目　录

第一章　软件工程概述

在近代技术发展的历史上，工程学科的进步一直是产业发展的巨大推动力。传统的工程学科走过的道路已为人们所熟知，水利工程、建筑工程、机械工程、电力工程等对于工农业、商业、交通业的影响是极为明显的。人类在认识和征服大自然的长征中继续前进，近年来人们开始对气象工程、生物工程、计算机工程等有了新的认识。然而，对于工程学这个家族中的另一新成员——软件工程却很不熟悉。其实，这并不是因为它的地位无关紧要，恰恰相反，它对软件产业的形成和发展起着决定性的推动作用。我们说它在计算机系统的发展和应用中至关重要，说它在人类进入信息化社会时成为新兴信息产业的支柱，绝不过分，更非无稽之谈。人们对软件工程不了解，其根本原因是对软件本身认识不清。本章将对软件的特点、软件工程的形成及软件生命期等概念给出简要的介绍，以期使读者从中得到对软件工程最起码的理解。

1.1　软件的特点及分类

“软件”这一名词 60 年代初从国外传来，当时许多人说不清它的确切含意。software 一词确是 soft 和 ware 两字组合而成。有人译为“软制品”，也有人译为“软体”。现在我们都统一称它为软件。对它的一种公认的解释为，软件由三部分组成：程序、数据和文档。即按事先设计的功能和性能要求执行的指令序列，使程序能正常加工信息所需要的数据以及描述程序操作及使用的资料。尽管这个说法并不是计算机软件的精确定义，然而却有助于让我们把它和扩充了含意的广义软件相区别。因为指明某一行业生产技术、管理

制度等的所谓广义软件，今天已经进入我们的社会生活。

为了能全面、正确地理解计算机和软件，有必要让我们来分析一下软件的特点。

（1）计算机软件是一种逻辑实体，而不是物理实体。因而它具有抽象性。这个特点使它和计算机硬件或是其它工程对象有着明显的差别。我们可以把它记录在纸面上，保存在计算机的存储器内部，也可以保留在磁盘、磁带上，但我们却无法看到软件本身的形态。

（2）计算机软件在研制、开发活动中被创造出来，但它不能按传统的“生产”含意加以理解。尽管软件开发和硬件制造之间也有着某些相似的步骤。但硬件制造过程中质量因素一直是至关重要的问题，而软件的情况却有很大的不同。软件的研制需要花费很大力气，一旦研制出来，大批生产几乎不花什么成本，是件非常容易做的事，其中的质量因素也比较容易掌握。正是由于这个特点，软件的复制太简单了。于是出现了软件的保护问题。为了使软件研制的复杂劳动受到社会的承认和尊重，必须从技术上和法律上采取有力的措施，对于任意复制软件的行为加以严格的限制。

虽然近年来国内外也都有建立“软件工厂”的说法，但软件工厂毕竟只是为软件开发手段或开发环境创造更加优越的条件，以利于高效地开发软件，并不意味着按硬件生产的模式生产软件。

（3）软件在长期运行和使用中没有磨损、老化、用旧等问题。任何机械、电子设备在运行和使用中，其失效率大都遵循 U 型曲线（即所谓“浴缸曲线”）。那是因为刚一投入使用时各部件尚未做到灵活运转，常常容易出问题。经一段运行，便可以稳定下来。而当设备已经历相当时期的运转，便会出现磨损、老化等问题，会使失效率突然提高。这意味着已经到达寿命的终点，即将报废了。软件情况与此不同，它没有 U 型曲线的右半翼，因为它不存在磨损和老化问题。然而软件在投入使用以后，可能要作修改，每次修改

都会引起失效率的提高(参看图 1.1)。事实上,软件在投入运行以后,由于发现错误,为适应运行环境或是需要对其功能加以扩充,都需要对软件进行修改。我们把这种修改称作软件维护。其实,这和硬件的维护有着本质的差别。

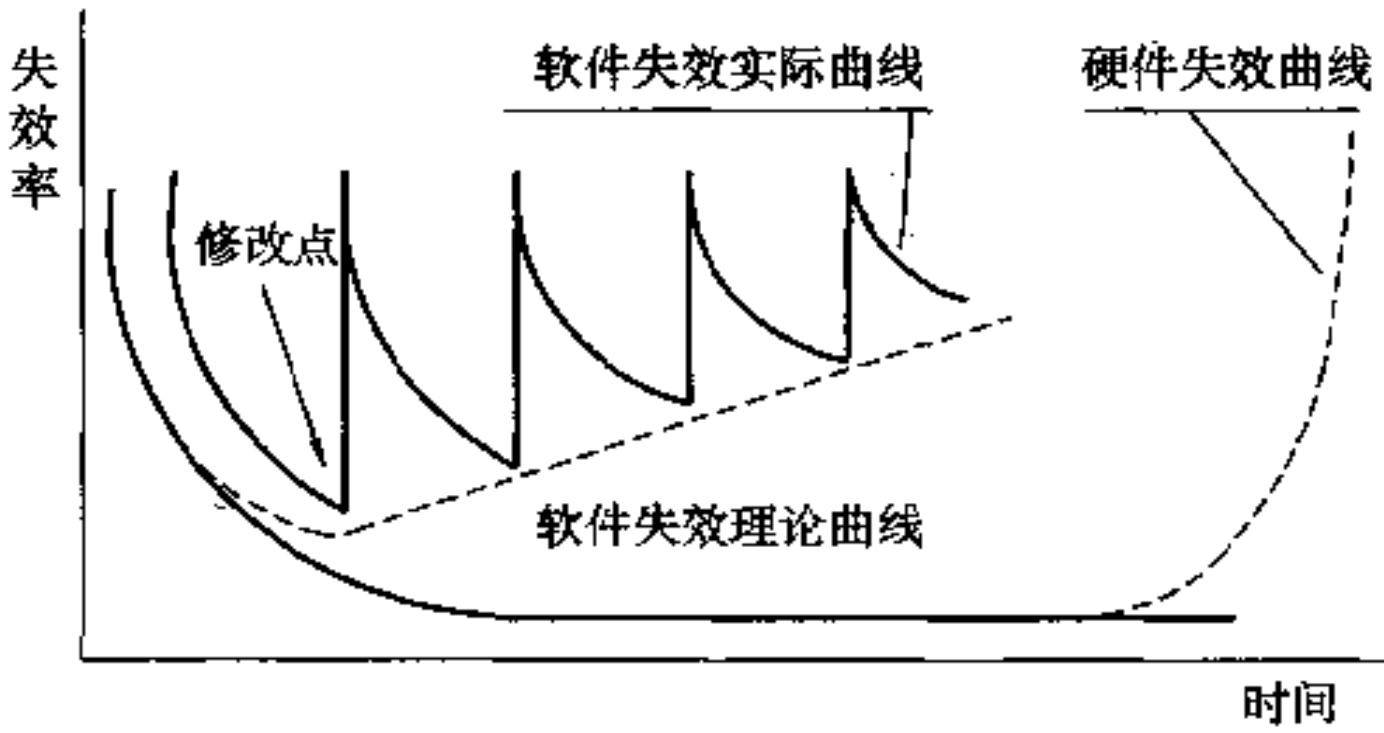


图 1.1　软硬件失效率曲线对比

(4) 软件的开发和运行常常受到计算机系统的限制,对计算机系统有着不同程度的依赖性。软件不能完全摆脱硬件单独活动。在开发和运行中必须以硬件提供的条件为依据。有的软件这种依赖性大些,常常为某个型号计算机所专用,这对使用将带来许多不方便。有的软件依赖于某个操作系统。为解除这种依赖性,在软件开发中提出了软件移植的问题。并且把软件的可移植性作为衡量软件质量的因素之一。

(5) 软件的开发至今尚未完全摆脱手工艺的开发方式。软件产品大多是"定做的",很少能做到利用现成的部件组装成所需的软件。近年来软件技术虽然取得了不少进展,提出了许多新的开发方法,比如充分利用现成软件的复用技术、自动生成技术,也研制了一些有效的软件开发工具或软件开发环境,但在软件项目中采用的比率仍然很低。传统的手工艺开发方式仍然占据统治地位。开

发的效率自然受到很大的限制。对于软件人员来说，开发工作是一种高强度的脑力劳动，没有那一个软件人员认为，这是一项轻松的工作。

(6) 软件本身是复杂的。有人认为，人类能够创造的最复杂的产物是计算机软件。软件的复杂性可能来自它所反映的实际问题的复杂性，比如，它所反映的自然规律，或是人类社会事务，都具有一定的复杂性；另一方面，也可能来自程序逻辑结构的复杂性。例如，一个系统软件要能处理各种可能出现的情况。软件开发，特别是应用软件的开发常常涉及到其它领域的专门知识，这对软件人员提出了很高的要求。软件的复杂性与软件技术的发展不相适应的状况越来越明显。图 1.2 示出软件技术的发展落后于复杂的软件需求，并且随着时间的推移，这个差距日益加大。

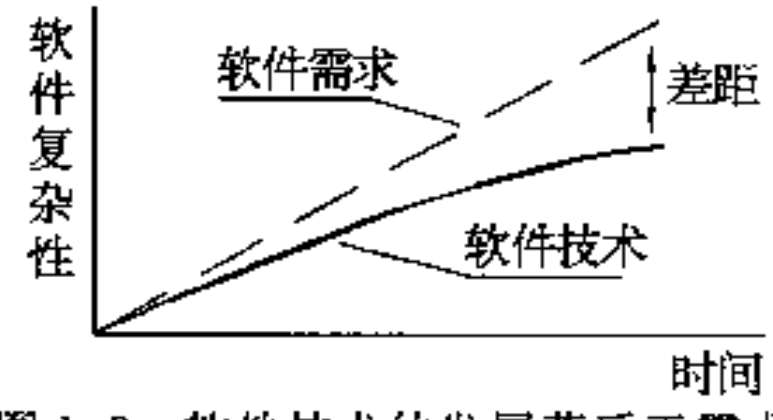


图 1.2　软件技术的发展落后于需求

(7) 软件是相当昂贵的。软件的研制工作需要投入大量的、复杂的、高强度脑力劳动，它的成本自然是较高的。问题不仅于此，值得注意的是硬软件的成本近三十年来发生了戏剧性的变化。无论研制也好，或是向厂家购买也好，在 50 年代末，软件的开销大约占总开销的百分之十几，大部分成本要花在硬件上；但今天这个比例要完全颠倒过来，软件的开销大大超过硬件的开销(参看图 1.3)。美国每年投入软件开发的经费要有几百亿美元。然而，也并非在所有软件开发上的花费都是成功的。

(8) 相当多的软件工作涉及到社会因素。类似于企业管理类

型的软件自然是不言而喻的。许多软件的开发和运行涉及机构、体制及管理方式等问题，甚至涉及到人们的观念和人们的心理。对于这些人的因素重视得不够，常常是软件工作遇到的问题之一。即使是对软件的看法不同也会有很大影响，比如，由于主管部门对正在开发的软件不够理解，因而得不到应有的重视和必要的支持，造成人力和资金上的困难，它直接影响到项目的成败。

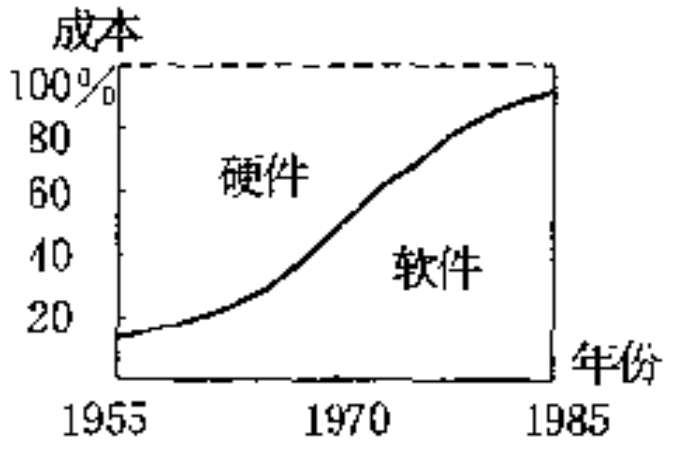


图 1.3 计算机系统硬、软件成本比例的变化

以上讨论的是区别于计算机硬件或是其它工程对象，各种软件的共同特点。究竟软件有哪些类型？事实上，要给计算机软件做出科学的分类是很难的，但鉴于不同类型的工程对象，对其进行开发和维护有着不同的要求和处理方法，因此仍然需要对软件的类型给出必要的划分。既然找不到一个统一的严格分类标准，我们从不同的方面去分类是比较符合实际情况的。

(1) 按软件的功能划分：一种通常的分法是三类，即系统软件、支撑软件和应用软件。

系统软件 能使计算机系统的各个部件、相关的软件和数据协调、高效地工作。例如，操作系统、数据库管理系统、驱动程序以及通讯处理程序等。系统软件的工作通常伴随着：频繁地与硬件来往、大量地为用户服务、资源的共享与复杂的进程管理，以及复杂数据结构的处理。系统软件是计算机系统必不可少的一个组成部分。

支撑软件 是协助用户开发软件的软件，包括帮助程序人员开发软件产品，也包括帮助管理人员控制开发的进程。表 1.1 给出了一些支撑软件的实例。

应用软件 能帮助人完成特定领域的工作。现在已经举不出

哪个国民经济部门完全不用计算机。为这些计算机应用领域服务的应用软件种类繁多。其中商用信息处理所占比例是最大的一类，工程和科学计算软件大多属于数值计算问题。值得一提的是，除去那些大量应用的传统领域以外，近年来一些新的应用领域如异军突起，十分引人注意。比如，计算机辅助设计(CAD)、系统模拟、智能产品嵌入软件(如汽车油耗控制、仪表盘数字显示，刹车系统)，以及人工智能软件(如专家系统、模式识别等)。应用软件在这些领域里大显神通，使得传统的产业部门面目一新，带给我们的是惊人的生产效率和巨大的经济效益。

表 1.1　支撑软件举例

一 般 类 型	支 持 需 求 分 析
文本编辑程序	PSL/PSA 问题描述语言、问题描述分析程序
文件格式化程序	关系数据库系统
磁盘向磁带作数据传输程序	一致性检验程序
程序库系统	CARA 计算机辅助需求分析
支 持 设 计	**支 持 实 现**
图形软件包	编译程序
结构化流程图绘图程序	交叉编辑程序
设计分析程序	预编译程序
程序结构图编辑程序	联接编辑程序
支 持 测 试	**支 持 管 理**
静态分析程序	PERT 进度计划评审方法绘图程序
符号执行程序	标准检验程序
模拟程序	库管理程序
测试覆盖检验程序	

（2）按软件的规模划分：按开发软件所需的人力、时间以及完成源程序的行数可确定六种不同规模的软件，即微型、小型、中型、大型、甚大型及极大型(参看第八章中表 8.1)。

（3）按软件工作方式划分：

·实时处理软件：在处理对象所发生不断变化的活动中，能随时配合完成量测、分析和控制的软件称为实时处理软件。对这样的软件来说，处理的时间是被严格限定的，如果在任何时间超出了这一限制，都将造成事故。

·交互式工作的软件接收用户给出的信息，不过时间上没有严格的限定。这种工作方式给用户很大的灵活。近年来，终端设备更加普及，交互式软件到处可见。一个重要的问题日益显得突出，这就是交互式软件的用户接口设计。良好的用户接口设计将给用户带来极大的方便。

·分时工作方式是按固定的时间段，轮流处理多个任务。

·批处理方式则是把多个任务一起送入机器，按顺序逐个处理的传统工作方式。

（4）按软件服务对象的范围划分：

有的软件只供一个(或少数几个)用户使用，可称这样的软件为项目软件(project software)。例如，军用防空指挥系统、卫星轨道控制系统的软件属于这一类。这类项目软件中有的软件，其开发目的带有试验研究性质。项目完成后可能需要在此基础上做进一步开发工作。

另一类则是要提供市场或是为千百个用户服务，我们称为产品软件(Product software)。一些通用的软件，如绘图软件包、编译程序、文本处理程序、报表生成程序等都属于这一类。由于要参与市场竞争，这类软件的功能、性能如何是很重要的因素。

（5）按软件的使用频度划分：

有的软件开发出来仅供一次使用，比如人口普查、工业普查所

需软件。若干年才进行一次普查，前些年开发的软件在若干年后很难适用。有的统计资料或试验数据需按年度做统计分析，相应的软件每年运行一次。另外一些问题，如天气预报，需每天进行数据的及时处理，相比之下，这类软件具有较高使用频度，显然，开发不同使用频度的软件，有不同的要求，不可一律看待。

(6) 按软件的失效影响划分：

工作在不同领域的软件，适应其不同的需求，在运行中对可靠性也有不同的要求。有的软件如果在工作中出现了故障，造成软件失效，可能给整个系统带来不大的影响。比如可能带来一些不便，却能勉强工作。但有的软件一旦失效，可能酿成灾难性后果，其严重损失难以挽回。比如控制载人飞行物的软件，如果不能正常工作，可能以人的生命为代价。开发这类软件自然应从多个方面采取措施，确保质量，做到万无一失。

1.2 软件的发展和软件危机

本世纪 40 年代中出现了世界上第一台计算机以后，就有了程序的概念。可以认为它是软件的前身。经历了 30 年的发展，使我们得以对软件有了更为深刻的认识。在 30 年中，计算机软件经历了三个发展阶段：

- 程序设计阶段，约为 50 至 60 年代
- 程序系统阶段，约为 60 至 70 年代
- 软件工程阶段，约为 70 至 80 年代

从表 1.2 中我们可以看到三个发展时期主要特征的对比。30 年来，最根本的变化体现在：

(1) 人们改变了对软件的看法。在 50 年代到 60 年代，程序设计曾经被看作是一种任人发挥创造才能的技术领域。当时人们认为，写出的程序只要能在计算机上得出正确的结果，程序的写法可以不受任何约束。一些程序尽管很难被别人看懂，但仍然认为，只

表 1.2　计算机软件发展的三个时期及其特点

特点 \ 时期	程序设计	程序系统	软件工程
软件所指	程　　序	程序及说明书	(项目软件) 产品软件
主要程序设计语言	汇编及机器语言	高级语言	高级语言系统 程序设计语言
软件工作范围	程序编写	包括了设计和测试	软件生存期
需求者	程序设计者本人	少数用户	市场用户
开发软件组织	个　　人	开发小组	开发小组 软件工厂
软件规模	小　　型	中小型	大中(小)型
决定软件质量的因素	个人程序技术	小组技术水平	管理水平
开发技术和手段	子程序 程序库	结构化程序设计	数据库、开发工具开发环境、工程化开发方法、标准和规范
维护责任者	程序设计者	开发小组	专职维护人员
硬件特征	价　高 存储容量小 工作可靠性差	降价，速度、容量及可靠性有明显提高	向超高速、大容量、及微型化发展
软件特征	完全不受重视	软件技术的发展不能满足需要，出现软件危机	开发技术有了前进，但未获突破性进展，价高 软件危机并未完全摆脱

有通篇充满了程序技巧，使用了许多窍门的程序才是高水平的好程序。然而，随着计算机的广泛使用，人们逐渐抛弃了这种观点。因为，对于小的程序，仅供极小范围使用（例如只是程序设计者本人或只有几个人），尚可“孤芳自赏”。对于稍大的程序，并需要较长时间为许多人使用的程序，情况就完全同了。人们要求这些程序要容易看懂、容易使用并且要容易修改和扩充。于是程序便从个人按自己意图创造的“艺术品”转变为能被广大用户接受的工程化产品。这时程序中难于理解的技巧成了有害的东西。

（2）软件的需求是软件发展的动力。早期的程序开发者只是为了满足自己的需要，这种自给自足的生产方式仍然是其低级阶段的表现。进入软件工程阶段以后，软件开发的成果具有社会属性，它要在市场中流通以满足广大用户的需要。软件开发者和用户的分工和责任也是十分清楚的。

（3）软件工作的范围从只考虑程序的编写到涉及整个软件生存期。关于软件生存期的概念将在下面一节介绍。

在软件技术发展的第二阶段，随着计算机硬件技术的进步，计算机的容量、速度和可靠性有明显提高，生产硬件的成本降低了。计算机价格的下跌为它的广泛应用创造了极好的条件。在这一形势下，要求软件能与之相适应。一些开发复杂的、大型的软件项目提了出来。然而软件技术的进步一直未能满足形势发展提出的要求。在软件开发中遇到的问题找不到解决的办法，致使问题积累起来，形成了日益尖锐的矛盾。这些问题归结起来有：

（1）由于缺乏软件开发的经验和关于软件开发数据的积累，使得开发工作的计划很难制定。主观盲目的制定计划，执行起来和实际情况有很大差距。致使经费预算常常突破。对于工作量估计不准确，进度计划无法遵循，开发工作完成的期限一拖再拖。已经拖延了的项目，为了加快进度赶上去而增加人力，结果适得其反，不仅未能加快，反面更加延误了。在这种情况下，软件开发的投资

者和软件的用户对软件开发工作既不满意，也不信任。

（2）作为软件设计依据的需求，在开发的初期阶段提得不够明确，或是未能得到确切的表达。开发工作开始后，软件人员和用户又未能及时交换意见，使得一些问题不能及时得到解决而隐藏起来，造成开发后期矛盾的集中暴露。然而这时问题既难于分析，也难于挽回了。

（3）开发过程没有统一的、公认的方法论或规范指导，参加的人员各行其事。加之不重视文字资料工作，使设计和实现过程的资料很不完整，或是每个人工作与其他人的接口部分被忽视。发现了问题修修补补，这样的软件极难维护。

（4）未能在测试阶段充分做好检测工作，提交用户的软件质量差，在运行中暴露出大量的问题。在应用领域工作的不可靠软件，轻者影响系统的正常工作，重者发生事故，甚至造成生命财产的重大损失。

这些矛盾表现在具体的软件开发项目上，最为突出的实例便是美国 IBM 公司在 1963 年至 1966 年开发的 IBM360 机的操作系统。这一项目花了 5000 人一年的工作量。最多时有一千人投入开发工作，写出了近一百万行源程序。尽管投入了这样多的人力和物力，得到的结果却是非常糟的。据统计，这个操作系统每次发行的新版本都是从前一版本中找出一千个程序错误而修正的结果。可以设想，这样的软件质量糟到什么地步。难怪这个项目的负责人 F. D. Brooks 事后总结了他在组织开发过程中的沉痛教训时说："……正像一支逃亡的野兽落到泥潭中作垂死挣扎，越是挣扎，陷得越深。最后无法逃脱灭顶的灾难，……程序设计工作正像这样一个泥潭，……一批批程序员被迫在泥潭中拼命挣扎，……谁也没有料到问题竟会陷入这样的困境……"。IBM360 操作系统的历史教训成为软件开发项目的典型事例为人们所记取。

以上这些矛盾多少描绘了软件危机的某些侧面，如果这些障

碍不能突破，进而摆脱困境，软件的发展是没有出路的。

1.3 软件开发工程化和软件生存期

从上述软件危机的现象和发生危机的原因可以看出，要想摆脱危机也不是很简单的事，不能只从一两个方面着手解决。真正得到满意的解决也并非一朝一夕能够做到的。但在这当中如何针对软件的特点，把它与其它产业部门工作对象的相同和相异之处加以分析，排除人们的一些传统观念和某些错误认识是非常重要的。只有端正了对软件的认识，真正抓住了它的特点和发展趋势才能逃出危机，走上软件发展的正确道路。

开发一个软件，除去那些规模很小的项目以外，通常要在多个软件人员的配合、协作之下共同完成；开发阶段之间的工作也应有很好的衔接；开发工作完成以后，软件成果要面向用户，在应用中接受用户的检验。所有这些活动都要求人们改变过去那种把软件当作个人才智产物的观点，抛弃那些只按自己工作习惯不顾与周围其它人员配合关系的做法。其实，在这一点上和研制计算机硬件，甚至和盖一座大楼又有什么本质的差别呢，任何参加这些工程项目的人员，他们的才能只有在工程项目的总体要求和技术规范的约束下才能得到充分发挥和施展。既然我们已经积累了几千年的工程学知识，能不能把它运用在软件开发工作中呢？实践表明，按工程化的原则和方法组织软件开发工作是有效的，也是摆脱软件危机的一个主要出路。

和工程学的概念对比，研究软件工作的特点进一步打开了我们的眼界。当我们全面分析软件各个“工序”的工作时，认识到程序编写只是软件工作的一个部分。在它的前后还有更重要的“工序”。正如任何其它事物一样，从发生、发展到成熟，以至衰亡，有一个历史发展的过程。计算机软件的生存期(Life Cycle)则包括：计划、需求分析、设计、程序编写、测试和运行维护六个步骤。这些步骤的主

要任务概括为：

• 计划(Planning)——确定要开发软件的总目标，给出它的功能、性能、可靠性以及接口等方面的设想。研究完成该项软件任务的可行性，探讨解决问题的方案。并且对可供使用的资源(如计算机硬、软件、人力等)、成本、可取得的效益和开发的进度作出估计。制定完成开发任务的实施计划(Implementation Plan)。

• 需求分析(Requirement Analysis)——对开发的软件进行详细的定义。这包括软件人员和用户共同讨论决定，哪些需求是可以满足的，并加以确切地描述。写出软件需求说明书(Software Requirement Specifications)或功能说明书(System Function Specifications)及初步的系统用户手册(System User's Manual)，提交管理机构评审。

• 软件设计(Software Design)——设计是软件工程的技术核心。在设计阶段中设计人员要把已确定了的各项需求转换成一个相应的体系结构，结构中每一组成部分是意义明确的模块，每个模块都和某些需求相对应，即所谓概要设计(Preliminary Design)。进而对每个模块要完成的工作进行具体的描述，以便为程序编写打下基础，即所谓详细设计(Detail Design)。所有设计中的考虑都应以设计说明书的形式加以详细描述，以供后继工作使用并提交审查。

• 程序编写(Coding，Programming)——把软件设计转换成计算机可以接受的程序，即写成以某一程序设计语言表示的“源程序清单”。这步工作也称为编码。自然，写出的程序应该是结构良好、清晰易读的，且与设计相一致的。

• 测试(Testing)——测试是保证软件质量的重要手段，其主要方式是在设计测试用例的基础上检验软件的各个组成部分。首先进行单元测试(Unit Testing)以发现模块在功能和结构方面的问题，其次将已测试过的模块组装起来进行组装测试，最后按所规定

的需求，逐项进行有效性测试，决定已开发的软件是否合格，能否交付给用户使用。

• 运行和维护(Run and Maintenance)——已交付的软件投入正式使用，便进入运行阶段。这阶段可能持续若干年甚至几十年。软件在运行中可能由于多方面的原因，需要对它进行修改。其原因可能有：运行中发现了软件中的错误需要修正；为了适应变化了的软件工作环境，需作适当变更；为增强软件的功能需作变更。

图 1.4 给出了软件生存期的瀑布模型。这个模型表明，在生存期中任何一个软件都要按顺序经历上述六个步骤，如同瀑布流水，逐级下落。然而，在工程实践中，为了确保软件产品的质量，每个步骤完成以后都要进行复查，如果发现了问题就应停止前进，沿着所经历的步骤返工。这就构成了图中各步骤间的向上流线。该图还指明了六个步骤可划分为三个阶段，即软件定义、软件开发和软件维护。

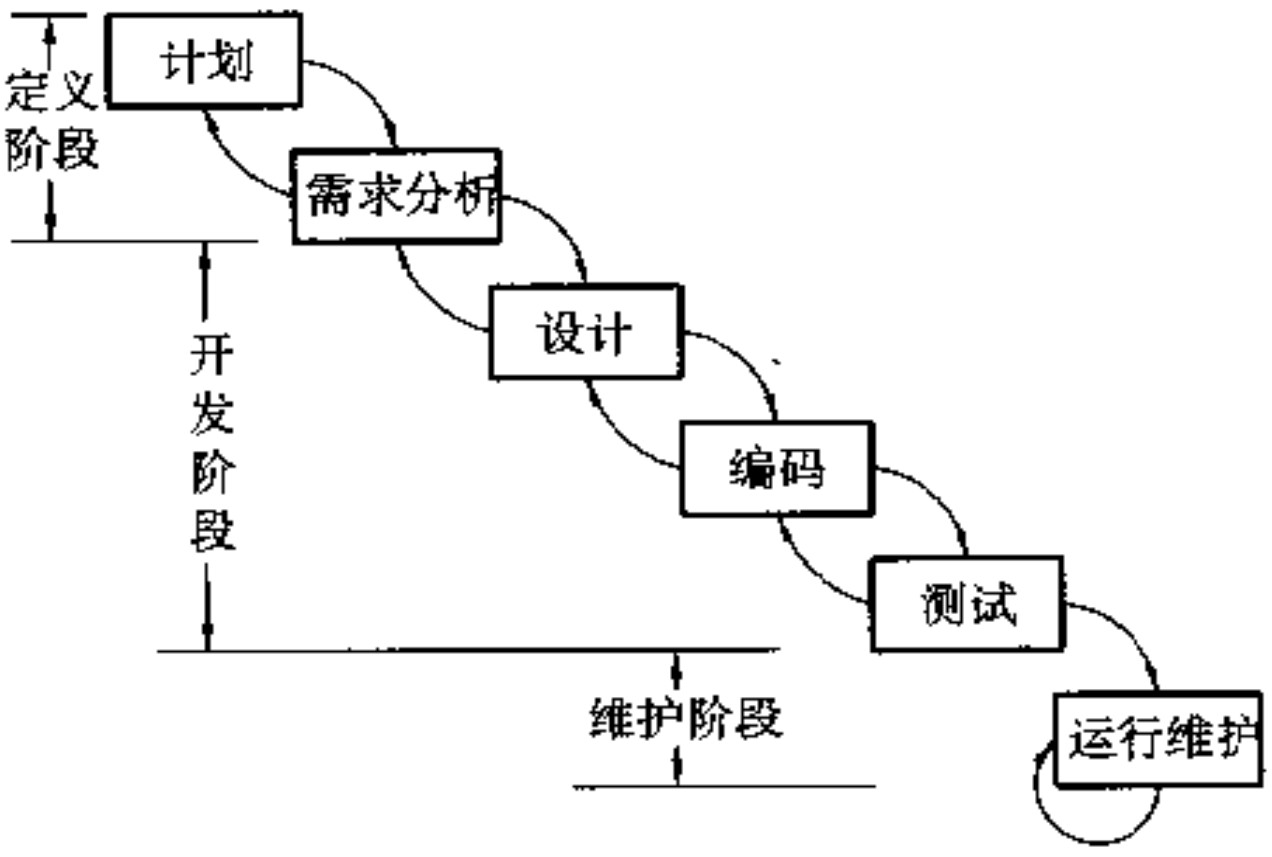


图 1.4 软件生存期的瀑布模型

软件维护在软件生存期中有着自己的特点。一方面，它是在软

件运行中提出要求的，例如，在运行中经过“评价”，确定修改的必要，则进入维护工作；另一方面，这时的修改工作仍然要经历生存期的各个步骤。因而，软件开发和维护各步骤的工作构成了生存期循环(如图 1.5 所示)。由于软件在它的运行过程中可能不只经历

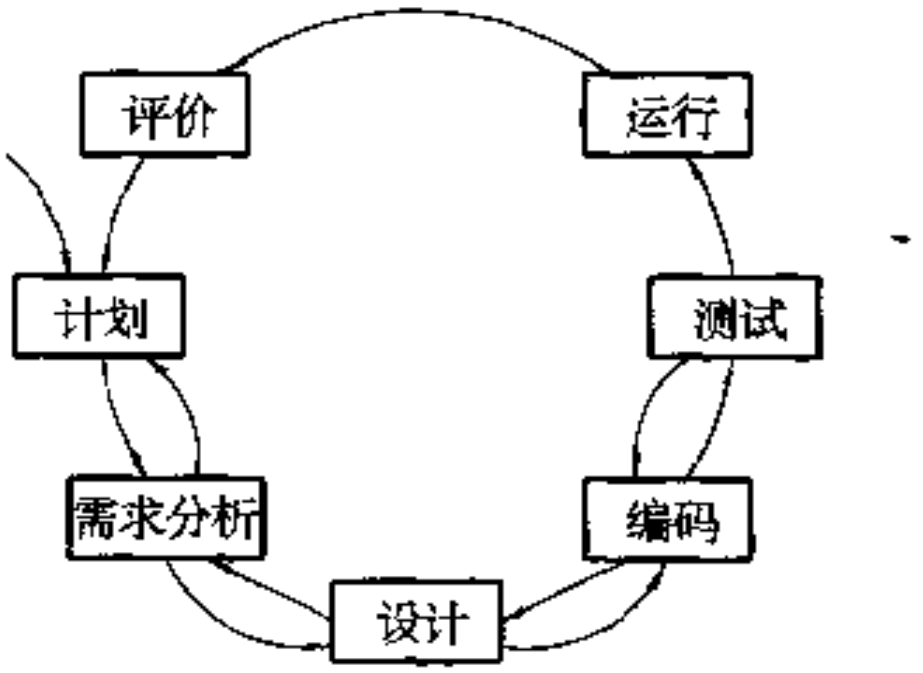


图 1.5　软件生存期循环

一次修改，为了把软件开发和维护工作所经历各个步骤区别开来，这里给出了 b 形的软件生存期表示法(图 1.6)。

由于开发软件沿着生存期要经历这么多步骤，开发出的软件产品能否真正符合要求，又只有到后期投入运行时才能得知，并且如果不能符合要求就将前功尽弃。为解决这一问题，近年来有人开始仿照硬件研制样机的办法开发软件，即研制软件样机(Prototyping)，也称建立快速原型。其方法是，给出初步的需求以后，用很短的时间建立一个具有基本功能的简单软件模型作为参考。经试运行，并和用户研讨后决定如何改进此模型或是按此雏型正式投入开发工作。实践表明，这是一种保证软件质量的有效开发方法。

为了更具体地说明软件生存期各阶段中的工作和应该完成的文档，在图 1.7、图 1.8 和图 1.9 中分别表示了软件定义阶段、开发阶段和维护阶段的各项活动。我们可从图中发现，各个阶段和步骤都安排了复审，其目的在于及时发现出现的问题，并加以解决，

不使其影响后面的工作。

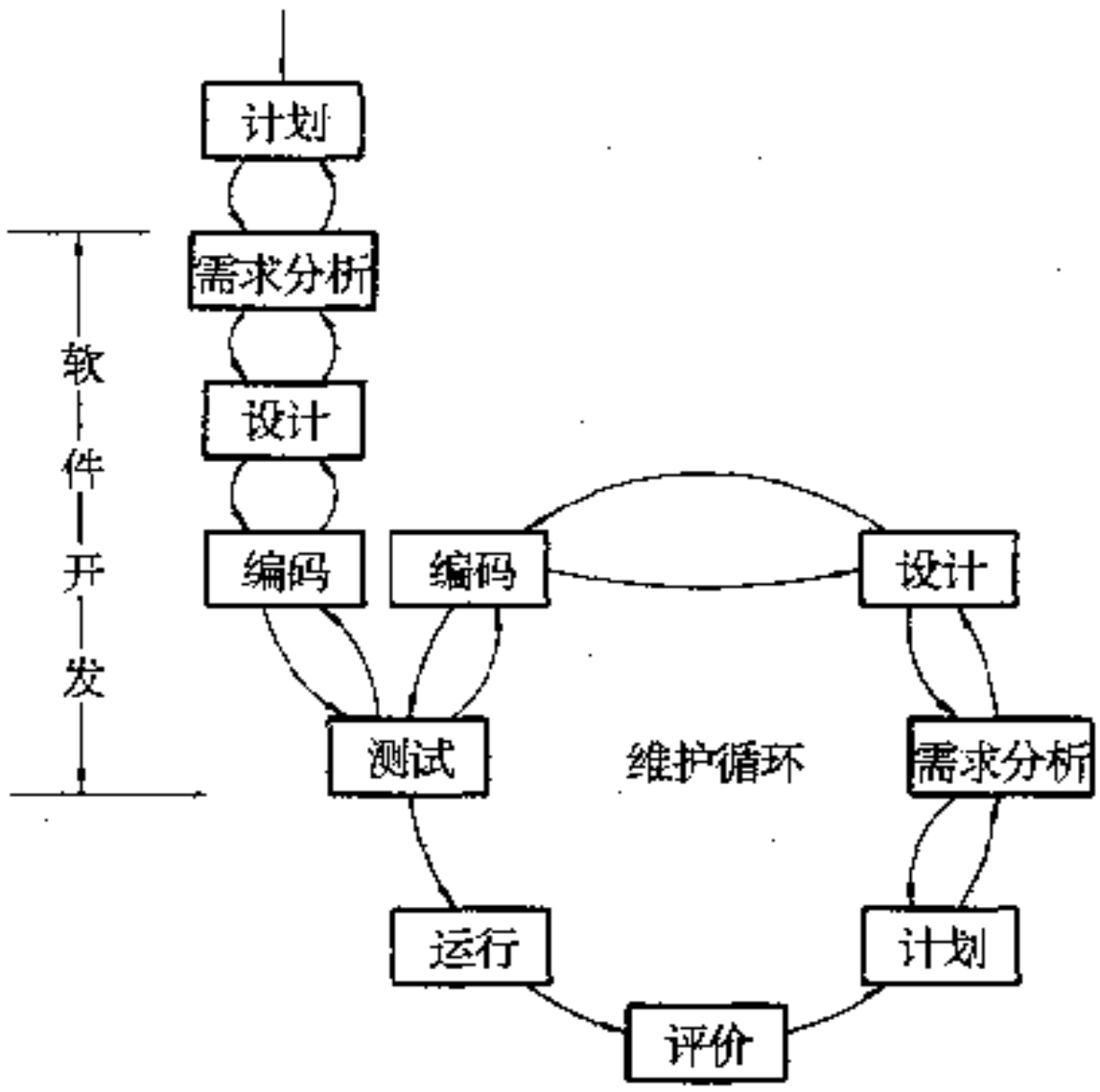


图 1.6　具有维护循环的软件生存期

1.4　软件工程项目的目标

组织实施软件工程项目，从技术上和管理上采取了多项措施以后，最终希望到得项目的成功。所谓成功指的是达到以下几个主要的目标：

- 付出较低的开发成本
- 达到要求的软件功能
- 取得较好的软件性能
- 开发的软件易于移植
- 需要较低的维护费用
- 能按时完成开发工作，及时交付使用

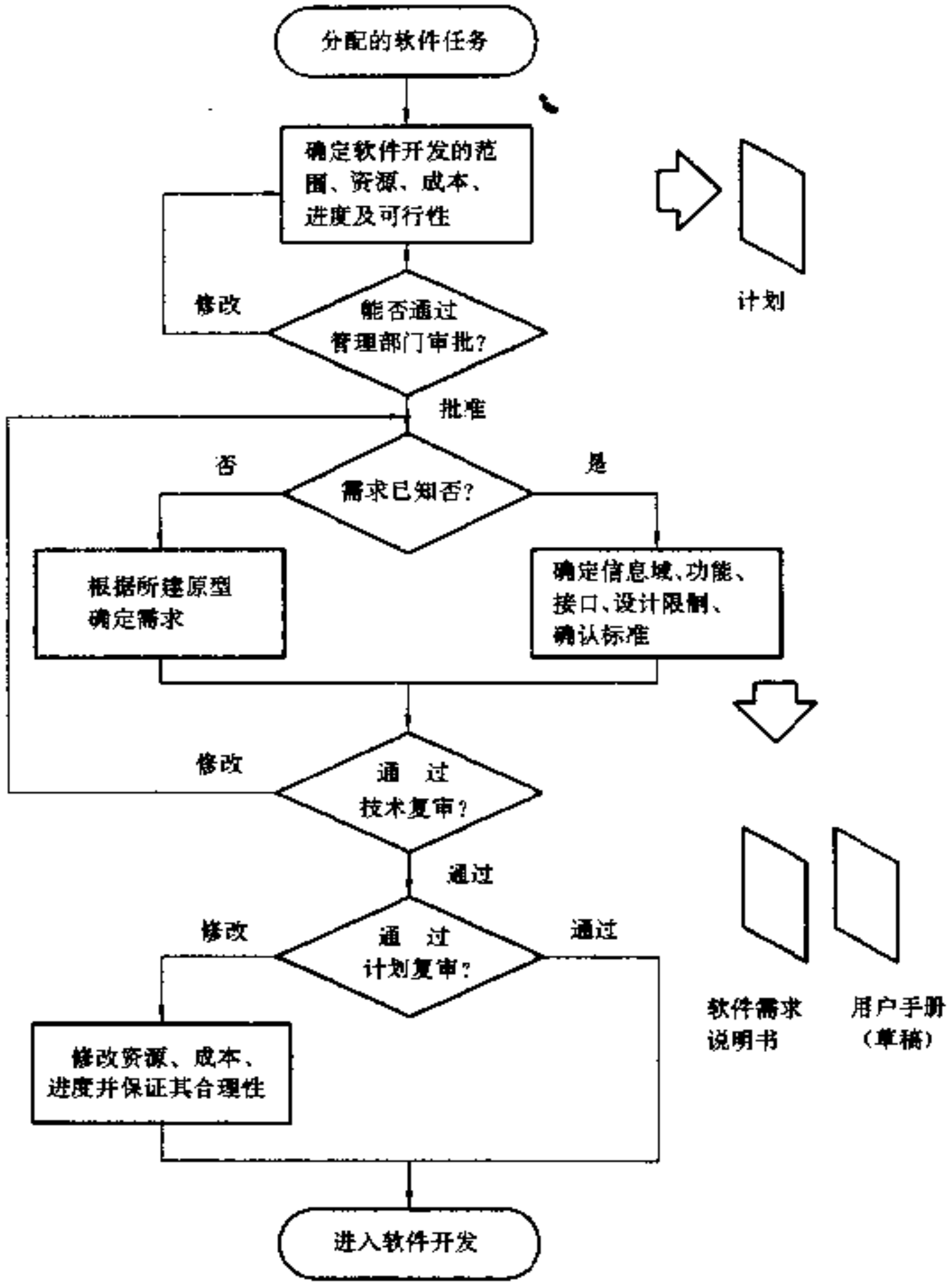

分配的软件任务
确定软件开发的范围、资源、成本、进度及可行性
修改
能否通过管理部门审批?
批准
否
需求已知否?
是
根据所建原型确定需求
确定信息域、功能、接口、设计限制、确认标准
修改
通 过技术复审?
通过
修改
通 过计划复审?
通过
修改资源、成本、进度并保证其合理性
进入软件开发
计划
软件需求说明书
用户手册（草稿）

图 1.7　定义阶段的工作

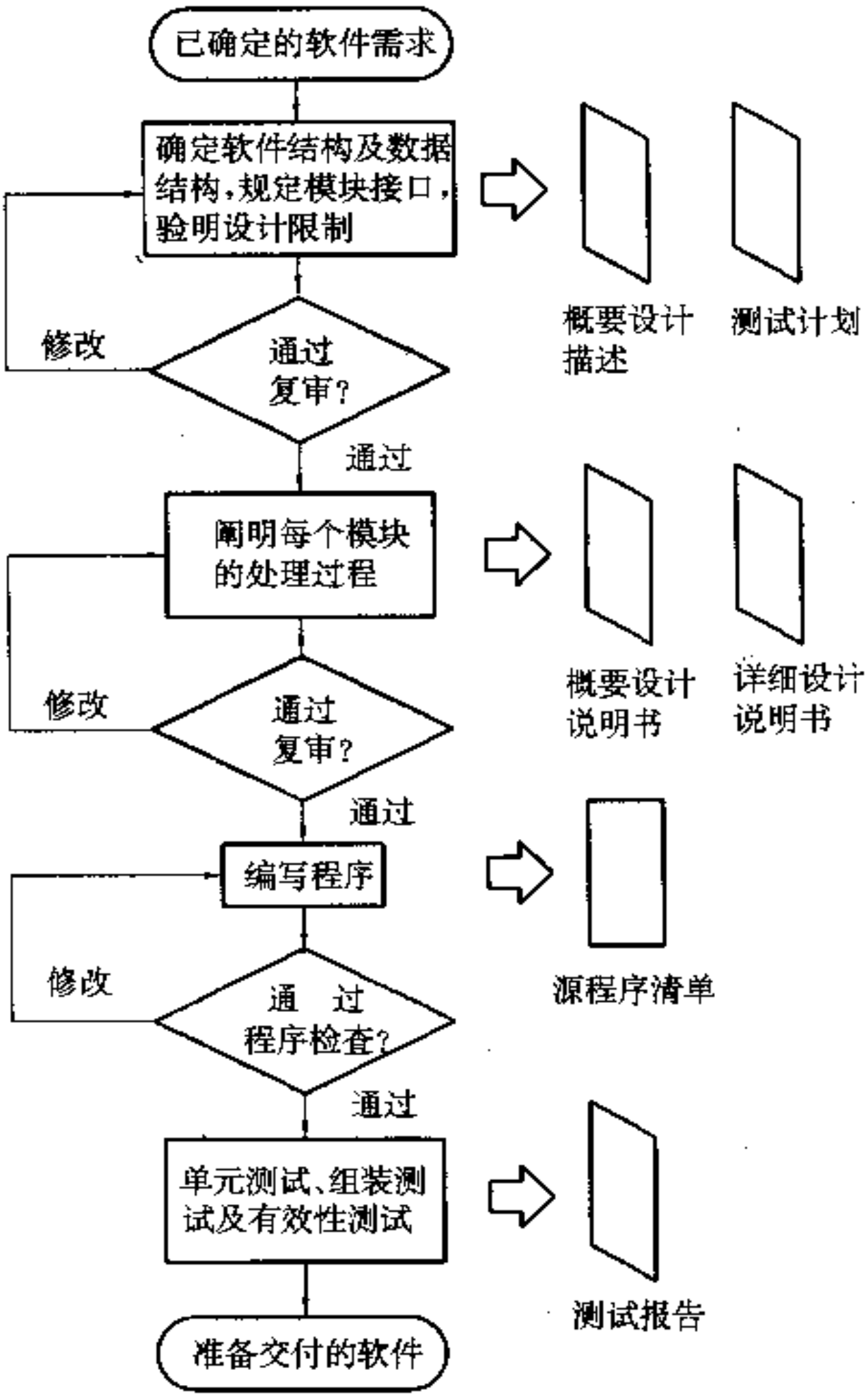

已确定的软件需求
确定软件结构及数据结构，规定模块接口，验明设计限制
修改
通过复审？
概要设计描述
测试计划
通过
阐明每个模块的处理过程
修改
通过复审？
概要设计说明书
详细设计说明书
通过
编写程序
修改
通 过程序检查？
源程序清单
通过
单元测试、组装测试及有效性测试
测试报告
准备交付的软件

图 1.8　开发阶段的工作

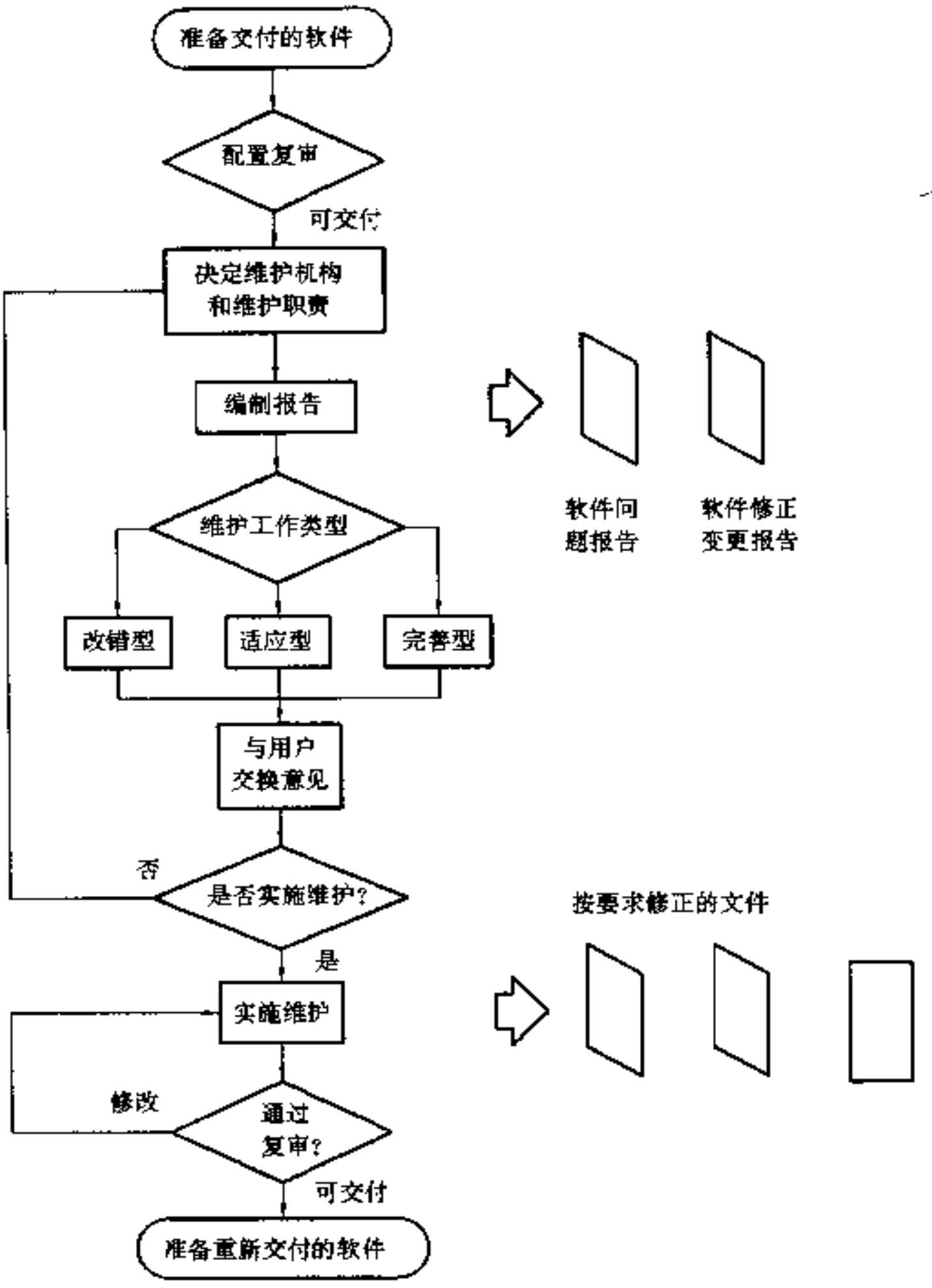


图 1.9　维护阶段的工作流程

实际上，在具体的项目中，企图让以上几个目标都达到理想的程度往往是非常困难的。而且上述目标很可能是互相冲突的。例如，假定只顾降低开发成本，很可能同时也降低了软件的可靠性。另一方面，开发工作中如果过于追求提高软件的性能，可能造成开发出的软件对硬件有较大的依赖，从而直接影响到软件的可移植性。

图 1.10 表明了软件工程目标之间存在的相互关系。其中有些目标之间是互补关系，例如，易于维护与高可靠性之间，低开发成本与按时交付之间。还有些目标是彼此与斥的，例如，上述指出的互相冲突的情况。

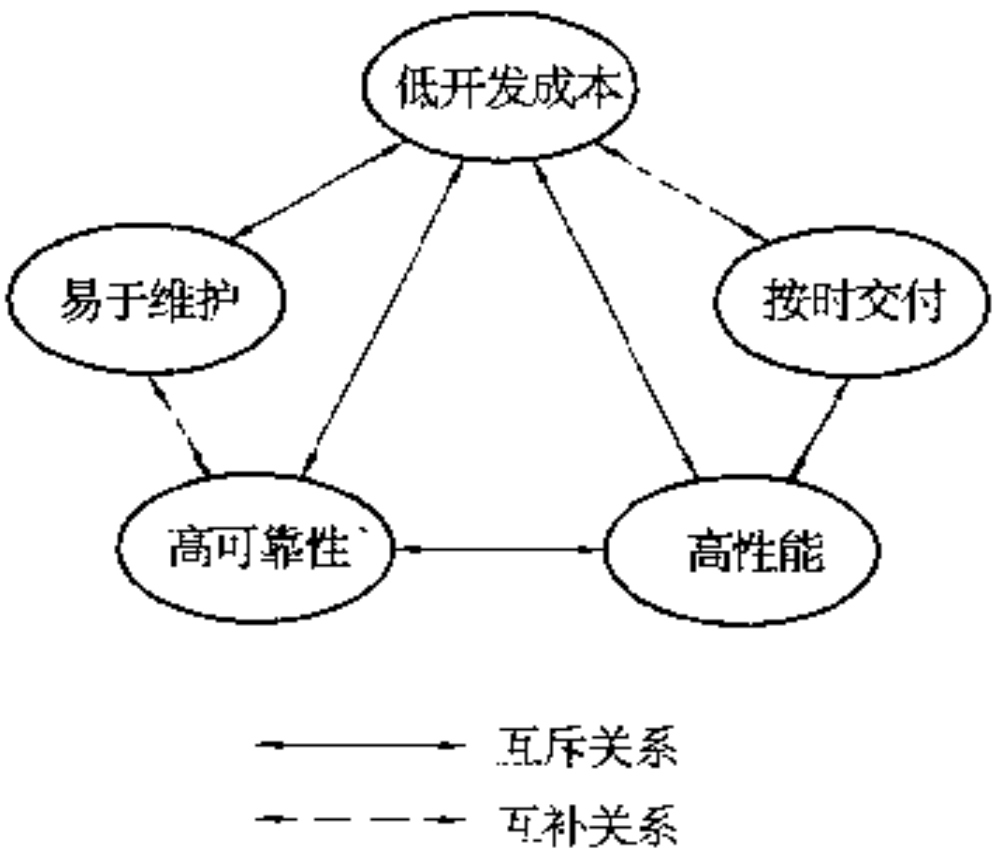


图 1.10　软件工程目标之间的关系

这里提到的几个目标很自然地成为判断软件开发方法或管理方法优劣的衡量尺度。如果提出一种新的开发方法，我们关心的是它对于满足哪些目标比现有的方法更为有利。实际上，实施软件开发项目就是力图在以上目标的冲突中取得一定程度的平衡。

第二章　软件需求分析

正如任何一件工作着手以前首先必须明确目标一样，软件开发工作在进行软件设计以前，应该弄清楚，要开发的软件应该具有哪些功能，应达到什么性能。明确了需求，就得到了软件设计的依据。表面看来，这个道理非常简单，也很容易做到。其实不然，软件开发的实践表明，做好需求分析并不是一件轻而易举的事。考察软件危机发生的原因之一便是忽视了需求分析这一重要步骤。往往是软件开发人员和用户未能全面地、准确地理解需求，或是未能恰当地表达这些需求，以致把需求分析阶段的遗留问题隐藏起来，并把它带到了开发工作的后期阶段，最终酿成不良的后果。

为了做好软件需求分析，需要了解需求分析阶段的任务，掌握需求分析的方法和工具。本章先重点介绍结构化分析方法，接着介绍面向数据结构的分析方法，最后引出结构化分析与设计方法。

作为软件开发的第一步，需求分析阶段的工作，应由分析员主持。经过了解用户的要求，认真细致地调研、分析，最终应建立目标系统的逻辑模型并写出软件规格说明书。以下从几个方面介绍需求分析阶段的工作。

2.1　需求分析阶段的任务

需求分析阶段研究的对象是软件产品的用户要求。需要注意的是，必须全面理解用户的各项要求，但又不能全盘接受所有的要求。这是因为并非所有用户提出的全部要求都是合理的。对其中模糊的要求还需要澄清，然后才能决定是否可以采纳。对于那些无法实现的要求应向用户做充分的解释，以求得谅解。

准确地表达被接受的用户要求是需求分析的另一重要方面。只有经过确切描述的要求才能成为设计的依据。

通常软件开发项目是要实现目标系统的物理模型。但是目标系统的具体物理模型是由它的逻辑模型经实例化得到的(参看图2.1)。作为目标系统的参考,当前系统可能是需要改进的某个已运行的数据处理系统,也可能是一个手工实现的数据处理过程。需求分析的任务便是借助于当前系统的逻辑模型导出目标系统的逻辑模型,解决目标系统"做什么"的问题。

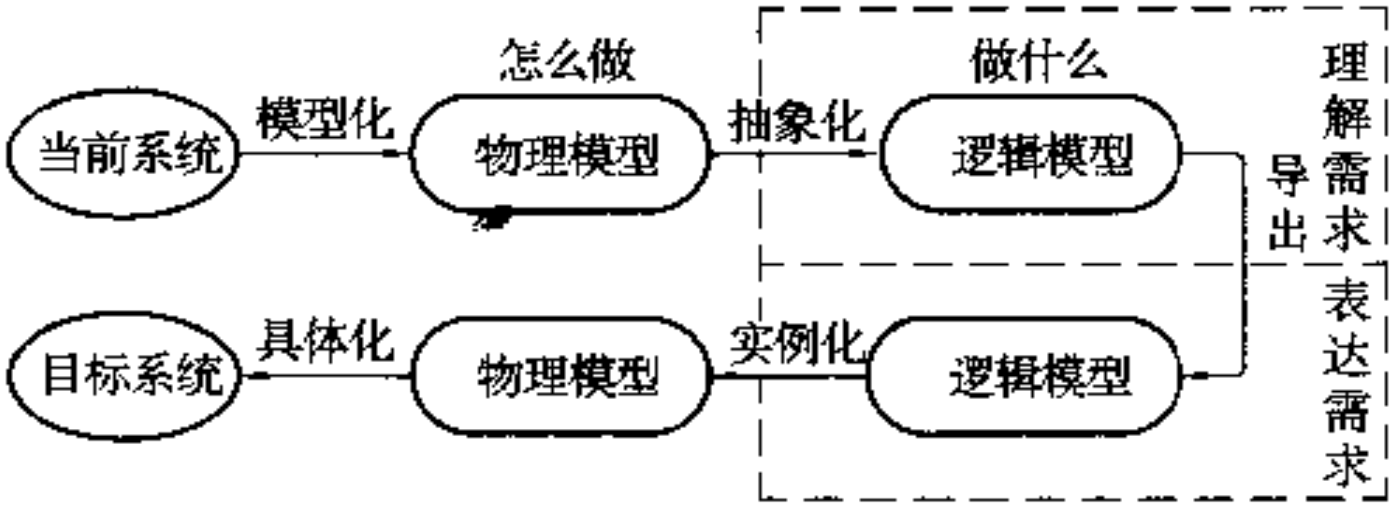


图 2.1 参考当前系统建立目标系统模型

需求分析阶段的工作概括地说,可分为四个阶段:

(1) 调查研究

调查研究是需求分析中掌握资料的基础工作。为了做好调查研究,应从以下方面着手:

·了解系统需求——软件开发常常是作为系统开发的一个组成部分。系统的需求分析工作自然直接涉及到软件的需求。因此,仔细研究系统分析的文档,了解对软件的要求,无疑是非常必要的一步。

·市场调查——了解市场上对要开发软件的需求形势,掌握市场上流通着的相关软件产品的技术和价格数据,对于决定开发的方针策略有着重要意义。在我国软件产业正在形成,市场经济日益起着重要作用的今天,掌握市场信息的意义逐渐被人们认识,它

正成为作出决策的重要依据。

·访问用户——从用户那里取得的信息常常有助于软件开发人员对系统文档的理解，并可通过与用户交换意见使得文档中的规定得到验证和澄清。此外，还可对文档提供的资料作进一步的补充。用户所提出的要求，应被当作重要的原始资料加以分析，即使当前无法实现的要求也应注意听取。

·考察现场——考察现场是直接掌握第一手资料的好方法。对工作现场的考察将有助于对软件所处理的信息流的分析，有助于对相关的硬件及其它接口部件的了解，特别是有助于对用户操作环境和操作要求的理解。使得软件开发人员更加确信哪些是对软件的基本要求，哪些是进一步要求，哪些要求可能在今后有变化，等等。

(2) 确定需求

确定需求就是要决定被开发的软件能够做什么，做到什么程度。或者说要决定软件开发人员要让它干什么，并且决定让它干得怎样。这些需求包括：

·功能需求——列举出被开发软件在职能上应做到什么。这是最主要的需求。

·性能需求——给出被开发软件工作的技术性能指标。比如，航空订票系统中的软件对于用户的订票请求，在几秒内作出回答。根据这些技术指标，用户可以对此订票系统作出评价。在具有相同功能的两个软件中，用户当然愿意使用响应时间短的一个，而不愿在键入订票的数据以后，在终端前多等一些时间。

·可靠性需求——第一章中我们已经了解到，各种软件在运行时，失效的影响各不相同。在需求分析中，应该对被 开发软件投入运行以后，不发生故障的概率，按实际的运行环境提出要求。对于那些重要的软件，或是运行失效会造成不良影响的软件应该提出较高的可靠性要求，以期在开发过程中采取必要的措施，使软件

产品能够高度可靠地稳定运行，从而避免运行事故带来的损失。

·安全和保密需求——工作在不同环境的软件对其安全、保密的要求显然是不同的。应该把这方面的需求恰当地作出规定，以便对被开发的软件给予特殊的设计，使其在运行中得到必要的保护。

·资源使用需求——这是指被开发软件运行时所需的数据、软件、内存空间等各项资源。

·开发费用和开发进展的需求。

功能性需求是人们普遍关注的，但常常忽视对非功能性需求的分析。其实非功能性需求并不是无关紧要的。它的主要特点是涉及到的方面多而广，因而容易被忽略。表 2.1 中简要列举了一些在完成软件需求分析时，应该考虑到的非功能性需求。很显然，任何一个软件的非功能需求都要根据它的类型和工作环境来确定。前一章里曾讨论过软件的分类，那里所列举的不同类型的软件，其非功能需求自然是有很大差别的。

需要说明的是，并不是用户提出的所有要求都被接受，需求的确定要有一个慎重选择的过程。这个过程中可行性研究起着重要作用。可行性研究的目的是仔细分析各种需求，充分估计所有的有利条件和不利条件，从而判断实现软件开发的目标是否可行。

可行性研究需从以下几个方面考虑：

·技术可行性——从技术上分析达到目标的可能性，检验有无重大技术障碍，阻碍软件开发项目的目标得以实现。

·经济可行性——在给定的人力、资金和时间范围内，达到预期目标是否现实，从经济上看是否值得。

·社会可行性——研究被开发软件的权利归属方面的问题，比如涉及到有软件版权的争议时，要考虑法律的有关规定以及可能带来的后果。

可行性研究的结果应当有两种：一种是肯定的，或者经过局部

表 2.1　软件的非功能性需求

目录系统的限制	性能	实时性
		其它的时间限制
		资源利用，特别是硬件配置限制
		精确度、质量要求
	可靠性	可用性
		完整性
	保密性	安全性
		保密性
	运行限制	使用频率、运行期限
		控制方式（如远程或局部地区）
		对操作员的要求
	物理限制	系统的尺寸、重量、电源、温度、湿度等限制
开发维护的限制	开发类型（实用型开发或试验型开发）	
	开发工作量估计，在采用具有实验型的累进开发法时，对资源，开发时间及交付的安排	
	开发方法	质量控制标准
		里程碑和评审
		验收标准
	优先性和可修改性	
	可维护性	

的需求修改仍然是肯定的；另一种则是否定的，得到的结论是不合算或是有重大的问题难于解决。因此，认为经过可行性研究只会得到肯定的结论是不符合实际的，或者可行性研究本身不够客观。若

是在这种情况下投入开发工作，必将给整个软件项目带来严重后果。据美国 IBM 公司的统计，经过市场调查所得资料的分析，进行可行性研究以后，只有 25 分之一的项目投入开发。由此可见，需求分析工作是严格的，它是决定软件项目命运的关键性工作。

③ 描述需求

已经确定下来的需求应该得到清晰、准确的描述。通常我们把描述需求的文件称为软件规格说明书或软件需求说明书。对于这个文件的作用、要求和内容将在本节的最后作详细的讨论。为了确切地表达用户对软件的输入、输出要求，需要编写初步用户手册，着重反映被开发软件的用户接口及用户使用的具体要求。

④ 需求分析复审

作为需求分析阶段工作的复查手段，在需求分析的最后一步，应该对功能的正确性、完整性和清晰性以及其它需求给予评价。例如，所确定的各项需求是否恰当，是否存在不一致的情况或是出现冗余，设计的限制是否合理，是否考虑过其它方案。复审还应注意审查需求分析阶段应完成的主要文档：软件需求规格说明书和初步用户手册是否符合要求。

为保证软件开发的质量，复审应以专门指定的人员负责，并按规范要求严格进行。复审结束应有复审负责人员的结论意见及签字。需要修改的部分，待完成修改以后才可进入设计阶段。

图 2.2 表示了需求分析阶段工作的大致步骤。

2.2 软件需求分析的原则

近年来已经提出了多种软件分析和说明方法，不过各种分析方法有着共同适用的基本原则：

(1) 要能够表达和理解问题的信息域以及功能域。

软件开发工作最终是为了解决好数据处理问题，就是要把一种形式的数据转换成另一形式的数据。其转换过程必定经历接收

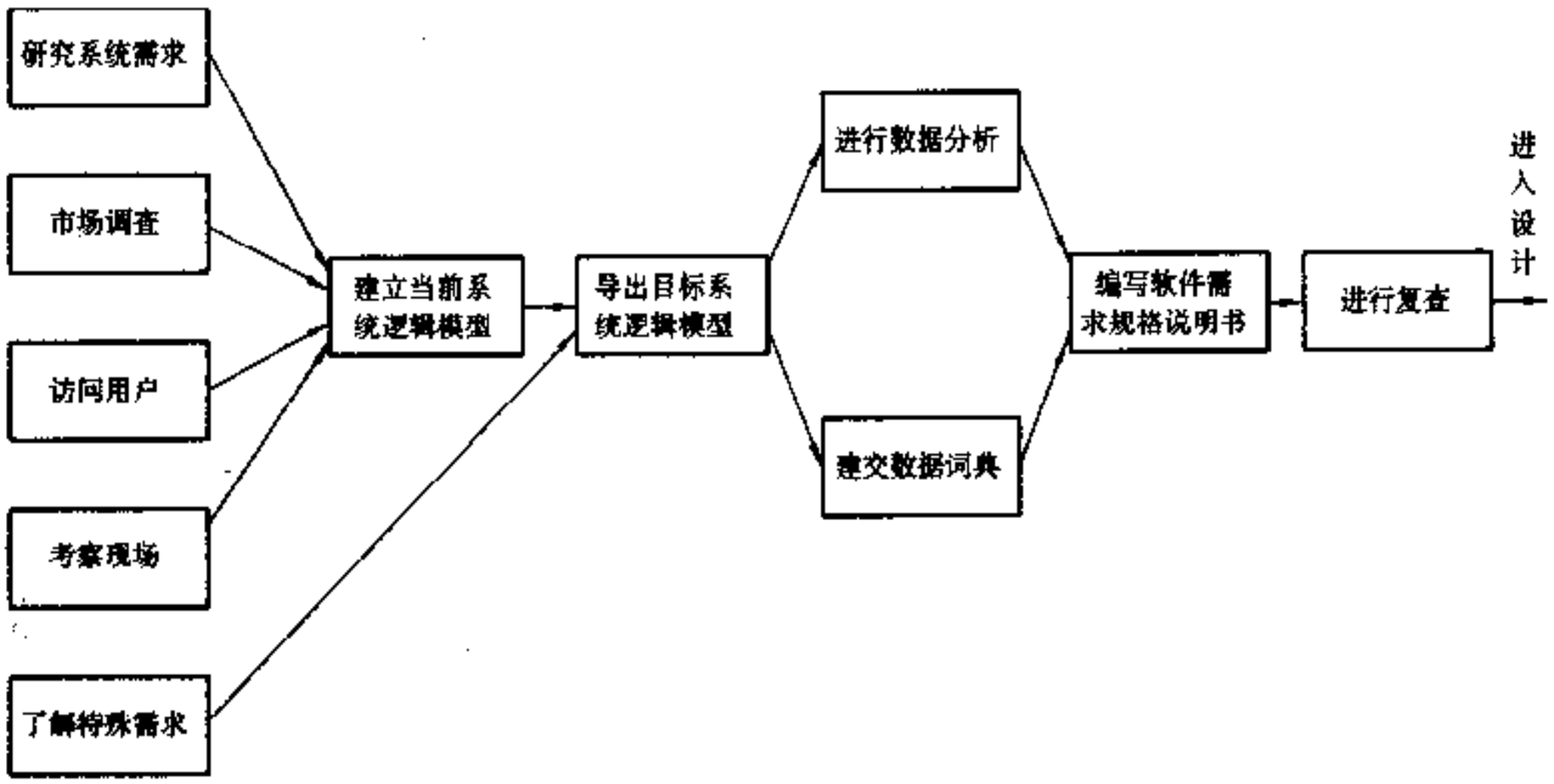


图 2.2　软件需求分析工作步骤

数据、加工数据和生成结果数据等步骤。所谓问题的信息域应包括：

- 被处理的信息流
- 信息内容
- 信息结构

(2) 问题应能以某种方式划分，使之按层次关系揭示问题的细节，从而把复杂问题化简。

在需求分析阶段，软件的功能域和信息域都可作进一步分解。这种分解可以是在同一层次上的，称为横向分解；也可以是多层次的纵向分解(参看图 2.3)。

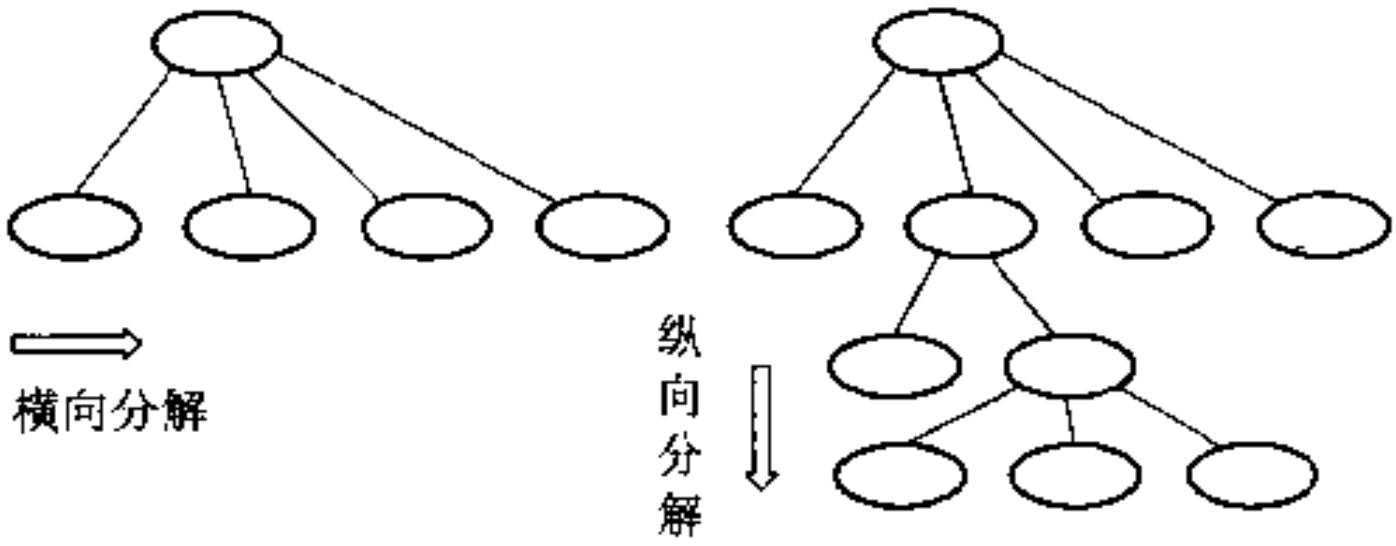


图 2.3　问题的分解

(3) 要给出系统的逻辑表示和物理表示，这对系统满足处理需求所提出的逻辑限制条件和系统中其它成分提出的物理限制是必不可少的。

软件需求的逻辑表示给出的是软件要达到的功能和要处理的信息，而不是实现的细节。物理表示给出的是处理功能和信息结构的实际表现形式，这往往是由设备本身决定的。比如一些软件靠终端键盘输入数据，但也有不少数据处理系统的软件靠模-数转换装置提供数据。分析员必须弄清它对软件的限制，并考虑功能和信息结构的物理表示。但这并不意味着要求分析员在需求分析中解决如何实现的具体问题。

2.3 分析员和用户的责任

软件需求分析的工作是软件开发人员和用户密切配合，充分交换意见，最终达到互相谅解的过程。作为开发人员一方的代表，参与需求分析的是分析员。他处在用户和高级程序员之间，负责沟通用户和开发人员的认识和见解，起着桥梁的作用。一方面要协助用户对所开发的软件阐明要求；另一方面还要和高级程序员交换意见，探讨用户所提要求的合理性以及实现的可能性。我们从图2.4中可以清楚地看出分析员在需求分析阶段起的重要作用。可以说他是需求分析的主要角色。

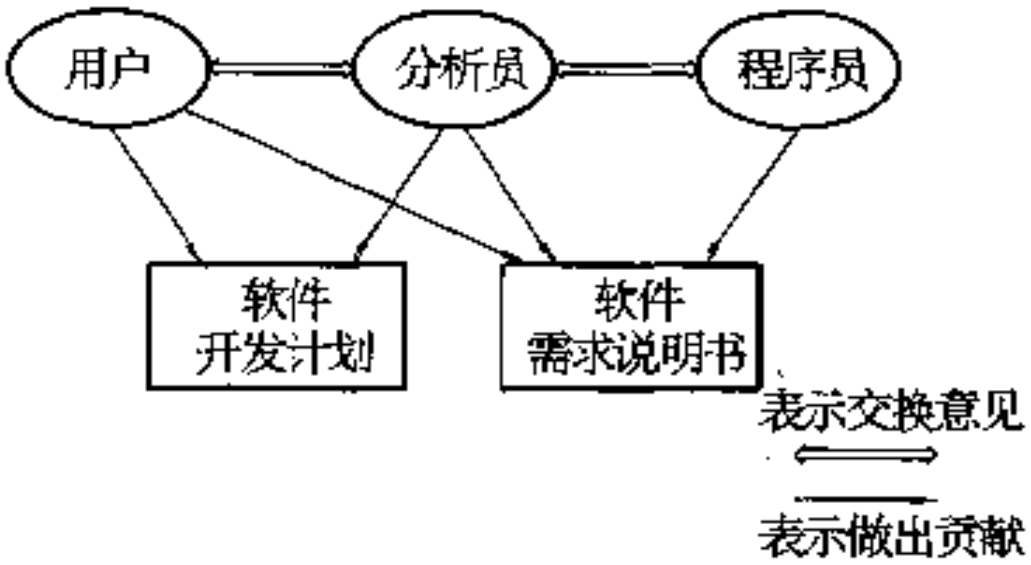


图 2.4 在软件需求分析阶段用户、分析员和程序员的工作配合

为能胜任上述任务，要求分析员熟练地掌握计算机硬、软件的专业知识，善于进行抽象的逻辑思维和创造性思维，能够倾听别人的意见，注意发挥其他人员的作用。分析员要负责编写软件需求说明书和初步用户手册。

用户在软件需求分析中也负有重要责任。这里“用户”不能简单地理解为某一个人。如果被开发的软件是在某个企业中运行的数据处理系统，所谓用户应该包括企业的业务负责人、企业中有关

部门的负责人以及与计算机系统运行有关的操作人员。“用户”应是他们的代表。这些人员在不同的工作岗位上分别熟悉并掌握着企业的技术发展方针、部门的业务工作以及相关的操作技术。他们对数据处理工作的考虑、他们对软件的意见和要求应该看做是需求分析十分宝贵的原始资料。分析员的工作应始终得到用户的密切配合。由此可看出，需求分析阶段软件开发人员应具有一定的人际交往能力，这对于保证软件满足用户的要求是必不可少的。

2.4 软件规格说明书

软件规格说明书是描述需求的重要文件，是软件需求分析工作的主要成果。它应着重反映软件的功能需求、性能需求、外部接口、数据流程等多个方面。不仅在开发过程中，而且在软件运行和维护的整个软件生命期中它都起着重要作用。

软件规格说明书中阐明的需求是经过认真研究和分析后肯定下来的。是软件开发人员和用户对问题的共同理解，可被当作是双方达成的协议书。由于其中规定的需求都是准备组织力量加以实现的，因此它应该作为软件设计和实现的基础和依据。在项目开发的最后阶段，其中规定的各项需求又将是产品验收的依据。当软件产品投入运行以后，如需进行适应性或扩充性修改，软件规格说明书仍然是十分必要的技术文件。

对于这样一个重要文件，应该具有以下几个方面的要求：

·正确性与完全性——按照不正确、不完全的需求开发出的软件产品不是用户所要的产品。

·一致性——不一致的规格说明在其各部分规定的需求是互相矛盾的。

·无多义性——多义性的需求将会使各人作出完全不同的解释。

·功能性——软件需求应具有功能的性质，它们应该清楚地

表明需要的是什么，无需表明系统将如何满足需求。这实际上是给软件设计人员很大的灵活性。

· 可验证性——一方面应能验证规定的需求能否满足用户的要求，另一方面还应能验证开发出的软件产品能否满足规定的需求。由于尚未找到正规的验证技术支持，目前最为重要的验证方法只是严格的逻辑推理。

· 可跟踪性和易于修改性——规定的需求应能检索、分割和交叉查找，以便于使用、修改和扩充。

此外，软件规格说明书应写得容易让用户看懂，特别应注意的是，要避免使用很多软件技术的专业术语。

2.5 结构化分析方法

结构化分析方法(Structured Analysis)简称 SA 方法，是面向数据流进行需求分析的方法。70 年代末经 Yourdon E.，Constantine L.，DeMarco T. 等人提出和发展，至今已得到广泛的应用。结构化分析的一些重要概念也包括在其它分析方法中，例如，结构化分析与设计方法(Structured Analysis and Design Technique—SADT) 和软件需求工程方法(Software Requirements Engineering Methodology—SREM)。

结构化分析方法适合于数据处理类型软件的需求分析。由于利用图形来表达需求，显得清晰、简明，避免了冗长、重复、难于阅读和修改等缺点，易于学习和掌握。近年来不仅得到广泛应用，而且一些软件开发机构开发了一些软件需求分析工具，支持这一方法，使之成为商品化的产品在市场上销售。

根据 DeMarco 的论述，结构化分析方法使用了以下几个工具：

· 判定表

· 判定树

· 数据流图

·数据词典

·结构化语言

其中数据流图用以表达系统内数据的运动情况。数据词典定义系统中的数据。结构化语言、判定表和判定树都是用以描述数据流的加工，下面将逐一介绍这些工具。

2.6 数据流图

数据流图简称 DFD，是 Data Flow Diagram 的缩写，也称为 Bubble Chart 或 Data Flow Graph，它是描述数据处理过程的有力工具。数据流图从数据传递和加工的角度，以图形的方式刻画数据处理系统的工作情况。作为一种描述手段它可以摸拟手工的、自动的或是两者混合的数据处理过程。

这里以我们熟悉的事务处理——去银行取款为例，说明数据流图如何描述数据处理过程。图 2.5 表示储户携带存折前去银行办理取款手续。他应把存折和填写好的取款单一并交给银行工作

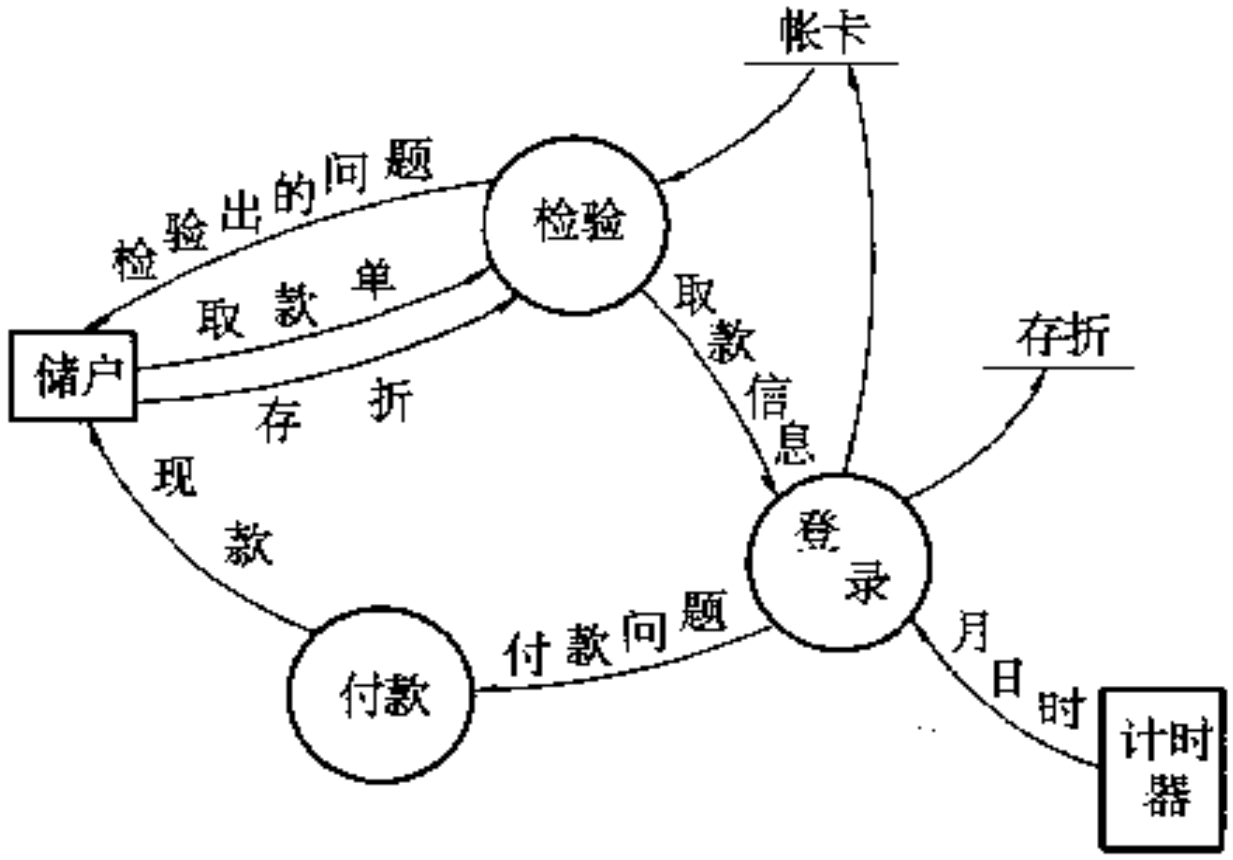


图 2.5 办理取款手续数据流图

人员检验。工作人员则需核对账目，发现存折有效性问题、取款单填写的问题或是存折、账卡与取款单不符等问题均应报告储户。在检验通过的情况下，则应将取款信息登记在存折和账卡上，并通知付款。根据付款通知对储户如数付款，从而完成这一简单的数据处理周期性活动。

从数据流图上我们看出，可能有四种基本成分出现：

① 数据流：图上常是命名的箭头。

② 加工：内有加工名的圆圈。

③ 文件：标有名字的短粗线。

④ 数据源点或数据终点：以方形框表示。

数据流是沿箭头指向传送数据的通道，它们大多是在加工之间传输被加工数据的命名通道。也有联接文件和加工的未命名的数据通道，即从加工指向文件或从文件指向加工的数据流，这些数据流虽未曾命名，因所连接的是有名加工和有名文件，所以其含意也是清楚的。

同一数据流图上不能有两个数据流同名。多个数据流可以指向一个加工，也可从某个加工散发出多个数据流。

加工是数据流图中的另一重要成分，它以数据结构或数据内容为加工对象。加工的名字常可分解为一个及物动词和一个名词作宾语，因而简明扼要地表明了完成的是什么加工。不过应注意，如果在加工命名时使用像“处理”、“控制”、“执行”、“调整”等不精确不具体的动词将会使得整个数据流图难于理解或是造成误解。

文件在数据流图中起着暂时保存数据的作用，所以也被称作数据存储(Data Store)，它可以是数据库、或任何形式的数据组织。指向文件的数据流可理解为写入文件，从文件引出的数据流理解为自文件读出。

数据流图上的第四种元素是数据源点或终点，它表示图中所

出现数据的始发点或终止点。由于它在图中的出现仅仅是一种符号，并不需要以软件的形式进行设计和实现，因而，原则上讲，它不属于数据流图的核心部分，只是数据流图的外围环境部分。在实际问题中它可能是人员、计算机外部设备或是传感装置。

在数据流图中，如果有两个以上数据流指向一个加工，或是从一个加工中引出两个以上数据流，这些数据流之间往往存在一定关系。为表达这些关系，在这些数据流的加工附近可以标上不同的记号。这里以对某一加工流入两个或流出两个数据流为例，说明其间符号的作用。所用符号及其含意在图 2.6 中给出。

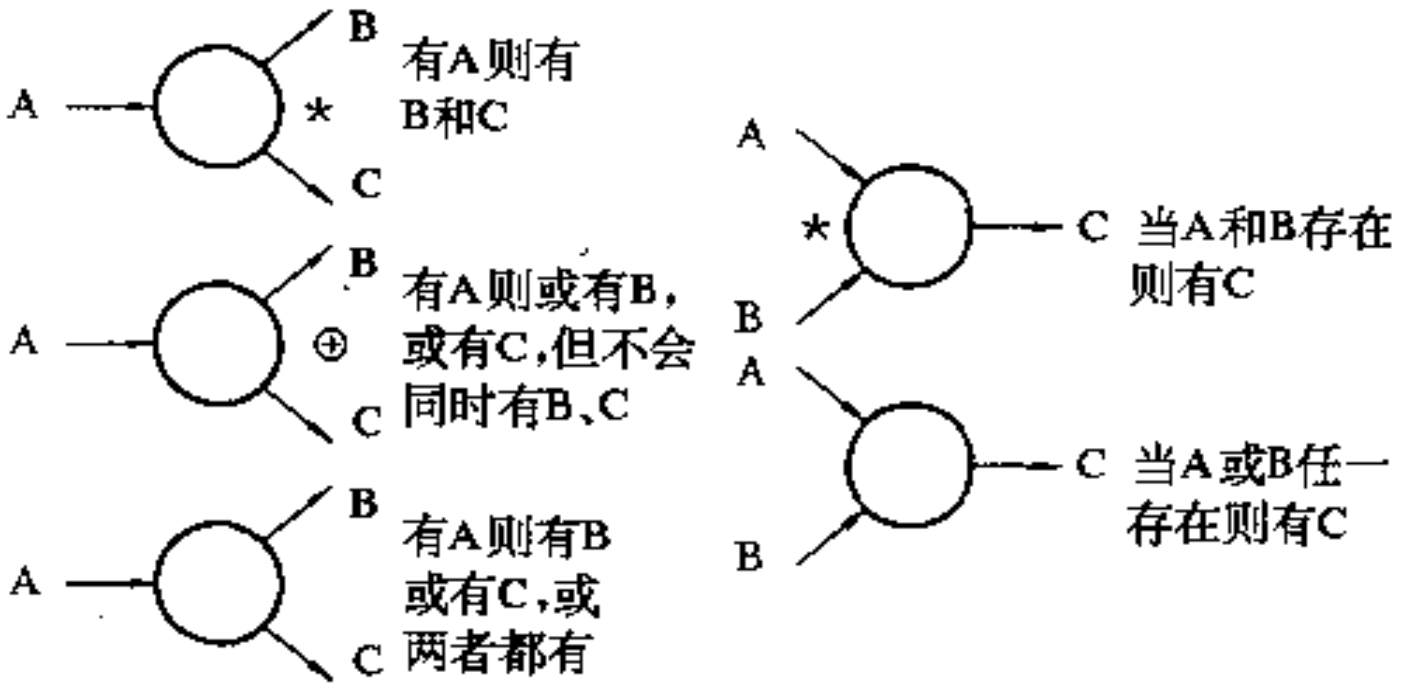


图 2.6　表明多个数据流与加工关系的符号

作为应用实例，图 2.7 给出使用上述符号的数据流图。这个实例的处理对象是旅客乘飞机前，在机场办理登机手续的数据处理系统。其主要步骤是，在正常情况下，旅客交验已购的机票和本人的身份证件及行李，按顺序经过四个数据加工：

- 票证检验——检验机票和证件的合法性、一致性，合格与否均给出信息。
- 行李安全检查——根据行李安全信息确认行李安全检查是否合格。

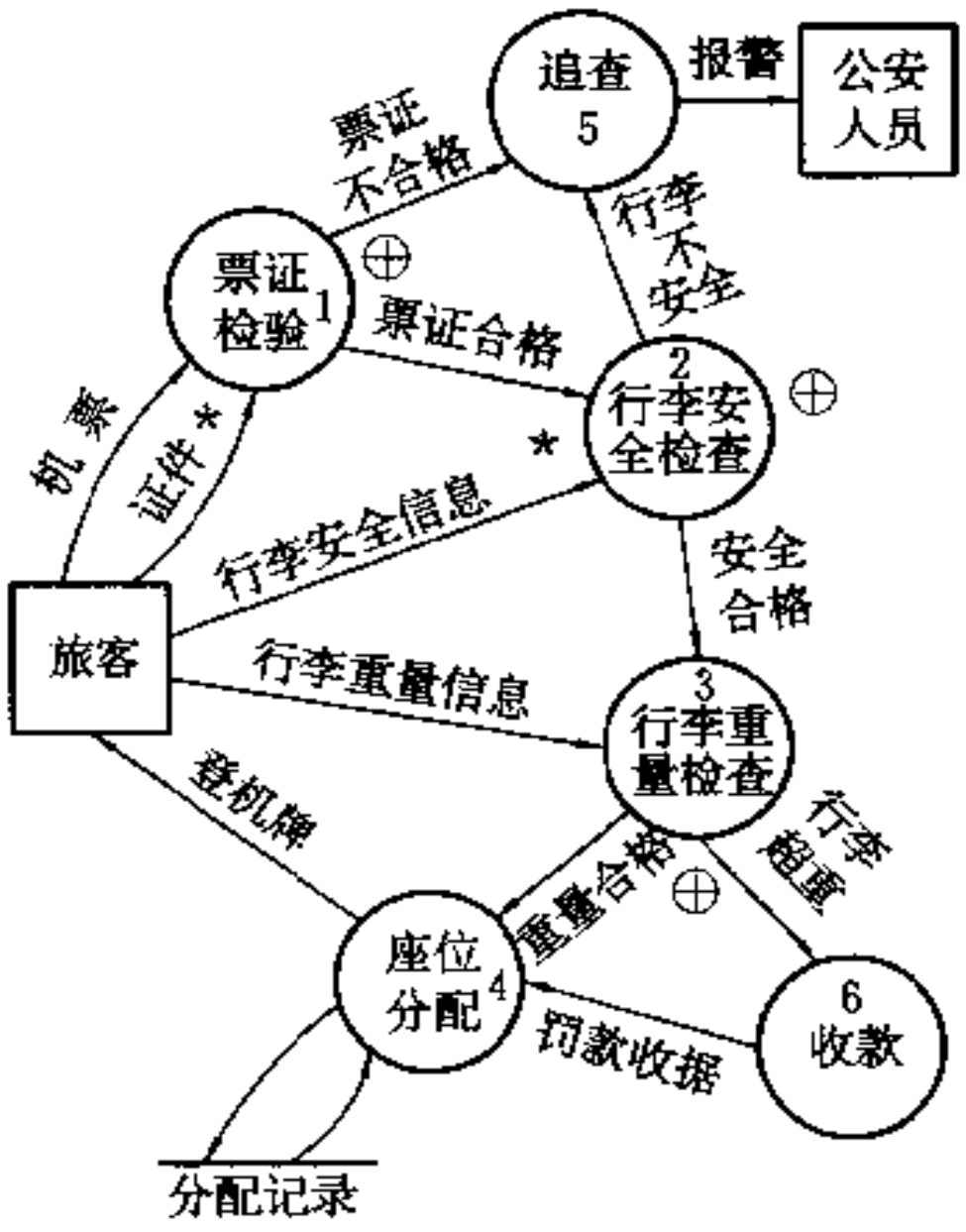


图 2.7　办理登机手续的数据流图

- 行李重量检查——由称出的行李重量得知是否超过规定携带行李重量。

- 座位分配——根据座位分配记录，给旅客分配空位后，将印有座位号的登机牌交给旅客，从而完成登机手续。

我们注意到，这四个加工的前三个都是检查性工作，指向这三个加工的输入数据流都必须同时到达，才能进行加工。因此在每个数据加工的两个输入数据流之间标有记号“*”。另一方面，这三个加工的输出数据流都是互为排斥的。如票证合格与不合格；安全检查通过与不通过；行李超重与行李重量合格。为表达这种互斥关系在每个加工的两个输出数据流间标以记号“⊕”。

标有这些记号的数据流图可以把数据流的加工情况描述得更

加确切，且读起来也更容易理解。

还应提醒读者特别注意的是，数据流图中的数据流是特定数据流向的表现形式，在程序流程图（也常常简称为程序框图）中箭头表示的控制流向有着本质的不同，绝不能混淆。熟悉程序流程图的初学者很容易把控制流的概念带到数据流图中来，这将造成混乱。例如，在程序流程图中某个逻辑变量取值为真，则执行 A，否则执行 B。这样的控制关系通常在数据流图中不予反映，而应在以后的设计和实现阶段加以解决。

为了表达数据处理过程的数据加工情况，用一个数据流图往往是不够的。稍为复杂的实际问题，在数据流图上常常出现十几个甚至几十个加工。这样的数据流图看起来很不清楚。层次结构的数据流图能很好地解决这个问题。按照系统的层次结构进行逐步分解，并以分层的数据流图反映这种结构关系，能清楚地表达和容易地理解整个系统。

我们可以把整个数据处理过程看成如图 2.8 那样的一个加工，它的输入数据和输出数据实际上反映了本系统与外界环境的接口。这就是分层数据流图的顶层。但仅此一图并未表明数据的加工要求，需要进一步细化。如果这个数据处理系统 S 包含三个子系统，就可画出表示三个子系统 1、2、3 的加工及其相关的数据流（参看图 2.9）。这便是第二层数据流图，我们记为 DFD/L1。继续分解三个子系统，从而可得出第三层数据流图 DFD/L2.1、DFD/L2.2 及 DFD/L2.3 分别是子系统 1、2 和 3 的细化。仅以 DFD/L2.2 为例，其中的四个加工编号均可联系到其上层图中的子系统 2。这样得到的多层数据流图可以十分清晰地表达整个数据加工系统的真实情况。对任何一层数据流图来说，我们称它上层图为其父图，在它下一层的图则称为子图。

还必须注意到，各层数据流图之间应保持“平衡关系”。例如，图 2.9 中的 DFD/L1 中子系统 3 有两个输入数据流和一个输出数

据流，那么它的子图 DFD/L2.3 也要有同样多个输入数据流和输出数据流，才能符合子图细化的实际情况。

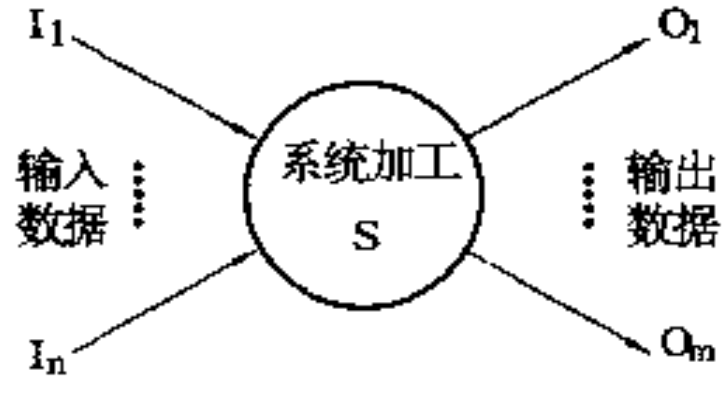


图 2.8

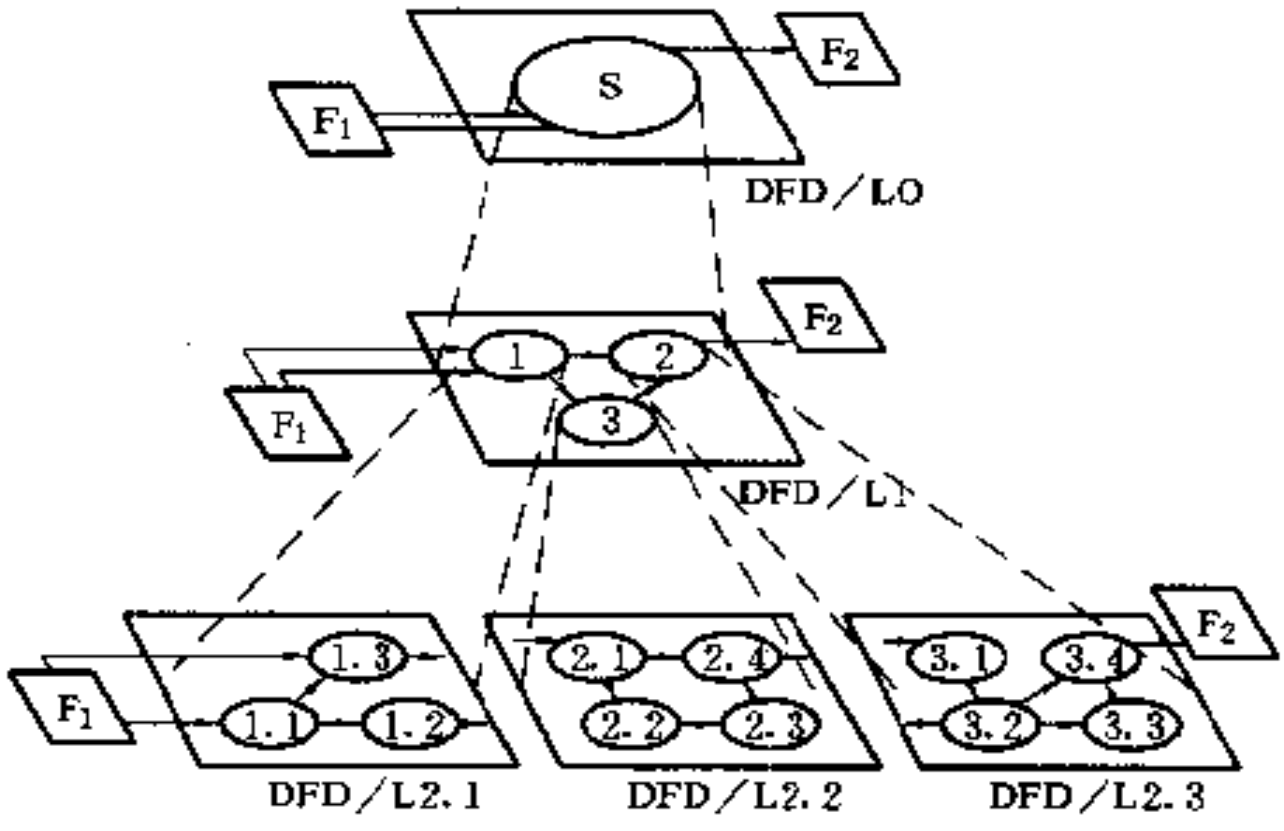


图 2.9

在多层数据流图中，我们可以把顶层流图、底层流图和中间层流图区分开来。顶层流图仅包含一个加工，它代表被开发的系统。它的输入流是待开发系统的输入数据，输出流是系统的输出数据。也许有人以为画顶层流图没有必要，因为它很简单。其实不然，顶层流图的作用在于表明被开发系统的范围，以及它和周围环境的数据交换关系。为逐层分解打下基础。而底层流图是指其加工不

需再作分解的数据流图，它处在最底层，有时也称其加工为“原子加工”。中间层流图则表示对其上层父图的细化。它的每一加工可能继续细化，形成子图。中间层次的多少视系统的复杂程度而决定。那么中间层流图细化到哪一层停止，才认为到达底层流图了呢？对于具体问题来说，答案并不唯一。DeMarco 认为，当一个加工的说明能在一页纸上容下时，层次的细化即可停止。有人则认为一个数据流图最多有七个加工，这样人们看起来不致因图上加工过多而眼花缭乱。这当然并没有什么理论根据，但却蕴含着人们的实践经验。

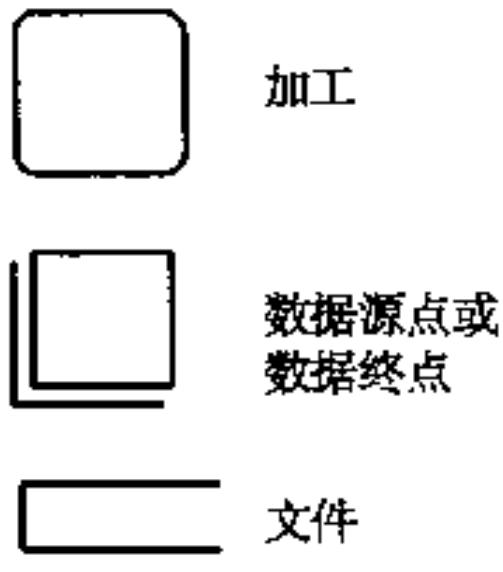


图 2.10

最后需要说明的是，为了使数据流图便于在计算机上输入输出，免去画斜线和圆的困难，常常使用另一套符号，如图 2.10 所示。

采用这组符号画出的取款数据流图和办理登机手续的数据流图在图 2.11 和图 2.12 中给出。

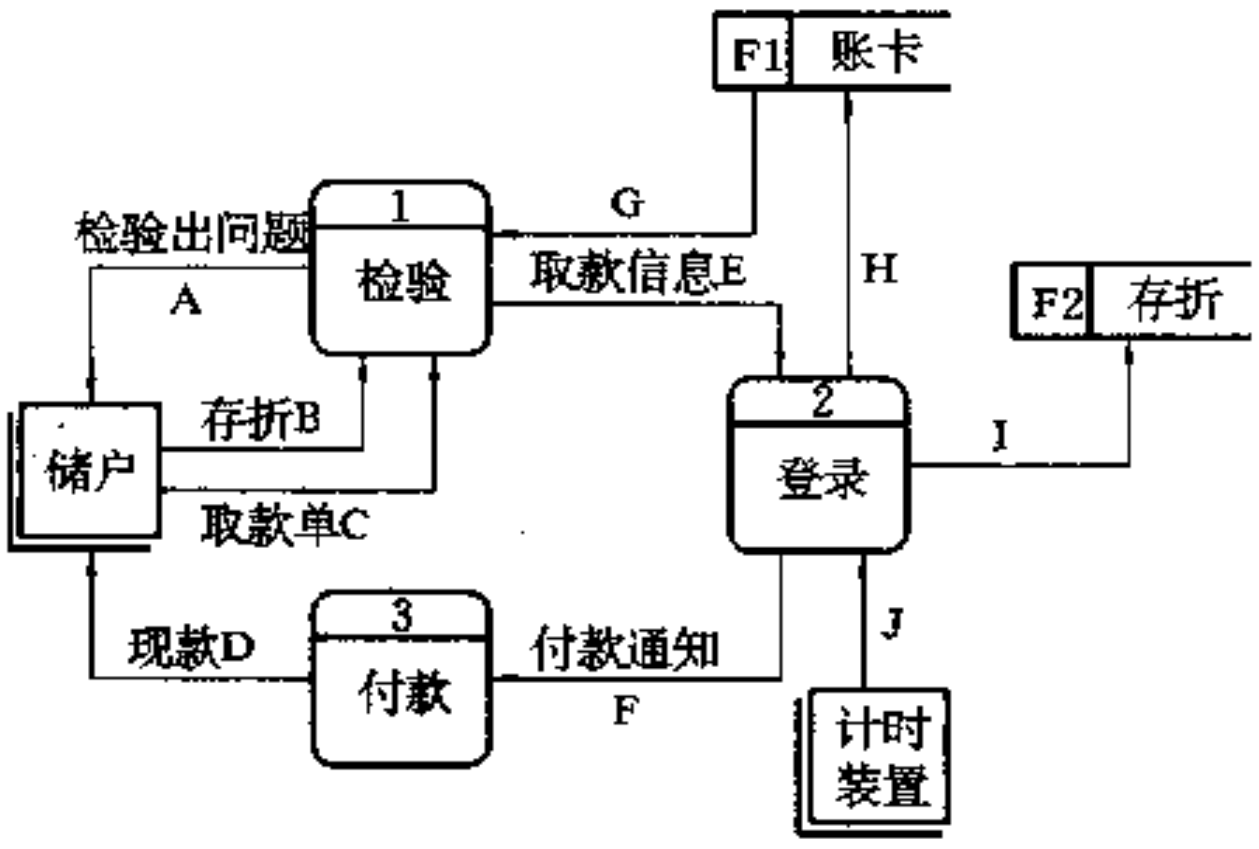


图 2.11

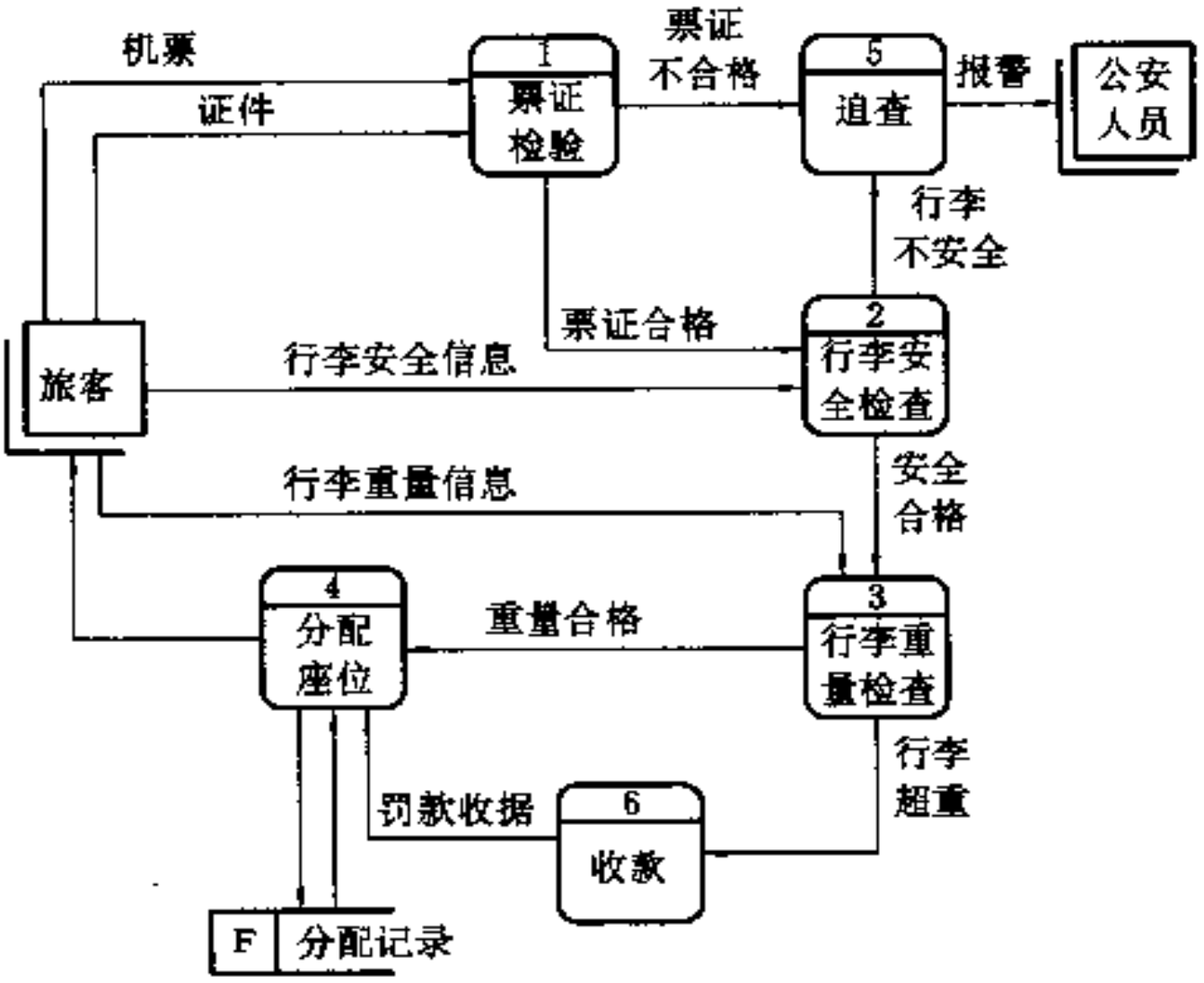


图 2.12

表 2.2　数据流图的等价矩阵

从 \ 到	储户	计时	F_1 帐卡	F_2 存折	检验	登录	付款
储　户	—				B,C		
计　时		—				J	
F_1 帐卡			—		G		
F_2 存折				—			
检　验	A				—	E	
登　录			H	I		—	F
付　款	D						—

为便于在计算机中存放数据流图，可将它表示成等价矩阵的形式。图 2.11 的等价矩阵如表 2.2 所示。其中，A、B、…、H 是数据流。

2.7 数据词典

数据词典 DD—Data Dictionary 是结构化分析方法的另一有力工具，它和数据流图密切配合，能清楚地表达数据处理的要求。数据流图给出了系统的组成及其相互的关系，但却未说明数据元素的含意。只有数据流图人们无法理解它所描述的对象。数据词典的任务是对数据流图中出现的所有数据元素给出定义。它使数据流图上的数据流名字、加工名字和文件名字具有确切的解释。所有名字按词条给出定义。全体定义式构成数据词典。

在图 2.5 表示的取款数据流图中，数据元素“存折”的格式如图 2.13 所示，它在数据词典中的定义式为：

存折＝帐号＋储户名＋{存取行}

这表明数据元素“存折”由三部分组成。第三部分的花括号表示“存取行”要重复出现多次。如果重复的次数是个常数，例如为 50，则可表示为 $\{存取行\}^{50}$ 或 {存取行}50

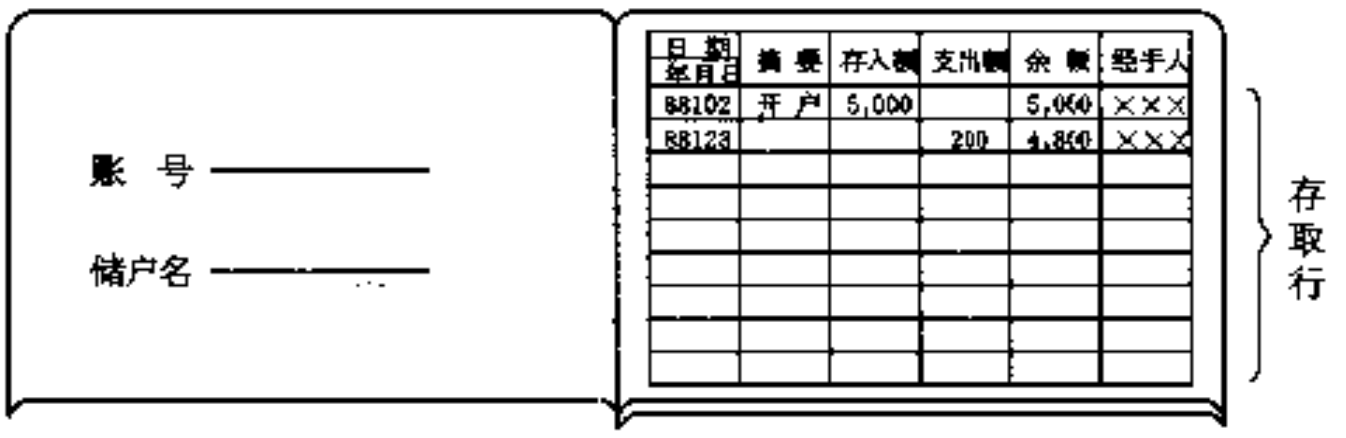


图 2.13 储户的存折格式

如果重复的次数是个变量，那么估计其变动范围，例如 10 至 50，

则可记为

{存取行}$^{50}_{10}$或 10{存取行}50

为进一步从上述三部分进行说明，需分别给出其定义式：

帐号＝“00001”..“99999”

储户名＝2{字母}24

存取行＝日期＋(摘要)＋存入额＋支出额＋余额＋经手人

这里给帐号规定了范围，指出它应是五位数的号码。以引号所给的具体数据取值已能完全确切地表明其含意，无需进一步定义，称为基本数据元素。但后两个定义式仍需进一步解释，即应对“字母”、“日期”、“摘要”等继续给出定义式。在“存储行”的定义式中“摘要”前后有圆括号，这表明，摘要是可有可无的。按照这样的方法，自顶向下，逐级给出定义式，直到最后出现无需定义的基本数据元素。

数据词典就是这样建立起来的一组定义式。必要时，有此定义式可能需要增加一些其它的解释行。同日常使用的词典一样，数据词典的定义式也要按一定顺序排列，如按字母顺序排列。当然不允许出现重复定义或是定义式相矛盾的情况。

通常在数据词典的定义式中出现的符号可能有以下几个：

符　号	含　义
＝	被定义为
＋	与
. .	连接符
[…,…]	或
[…|…]	或
{…}	花括号内多次重复出现
(…)	圆括号内可出现也可不出现
“…”	引号内给出的是基本数据元素

若 X、a 和 b 都是数据元素、以下定义式的含义分别在其右端给出：

X=a+b	X 是由 a 和 b 构成
X=〔a,b〕	X 是由 a 或 b 构成
X=〔a|b〕	X 是由 a 或 b 构成
X=(a)	a 可在 X 中出现，也可能不出现
X={a}	X 由零次或多次重复的 a 构成
X=m{a}n	X 由 m 至 n 个 a 组成，即至少有 m 个 a，至多有 n 个 a
X=a・・b	X 可取 a 至 b 的任一值
X="a"	X 为取值 a 的基本数据元素，即 a 无需进一步定义

这里再举一定义式实例。电话号码是一个三位至七位的十进制数，有的电话号码还需包括四位的分机号。它可取这样的定义式：

电话号码=3{十进数码}7(+"—"+分机号)

其中

十进数码="0"．．"9"

分机号=4{十进数码}4

这样，诸如

114、2467、32152、282451-2373 及 2561144 都是定义了的电话号码。

2.8 数据流图和数据词典应用实例

图书馆接待读者办理借书手续包括：验证(检验借书证的合法性)、验单(检验借书单填写的正确性)、找书和借书登记几个步骤。若各步骤之间均以特定的信息形式联系，让我们为这项借书业务(不包括还书、续借等)设计数据流图和数据词典。

(1) 数据流图

假定我们把检验证单(借书证与借书单)、检索存书和借书登记作为三个加工内容,并且设置库存书目和借书登记卡作为两个数据存储的文件。图 2.14 给出了其数据流图。

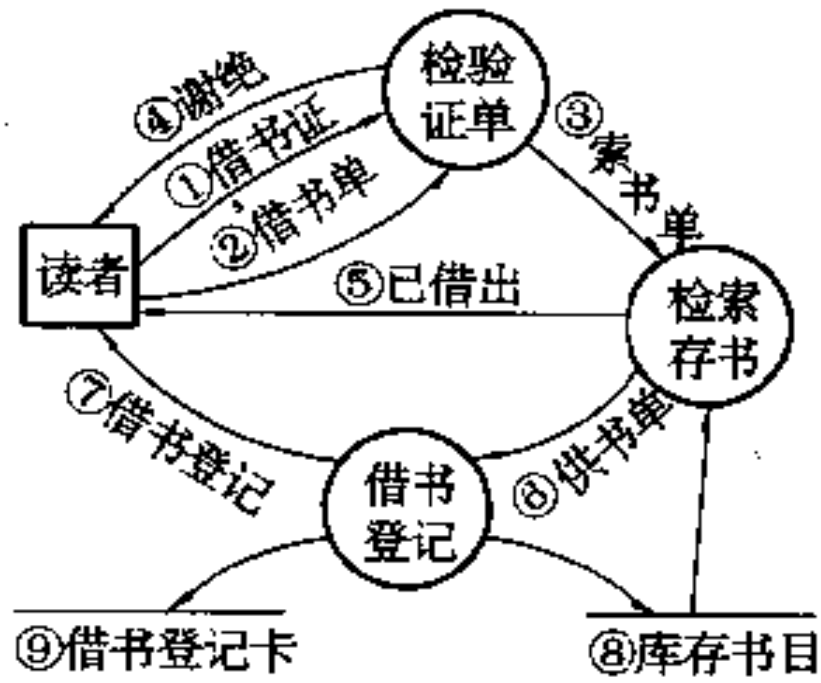


图 2.14 数据流图应用实例

尽管数据流图上给出了数据加工的情况,但一些命名的数据项,其含义仍不清楚,必须与数据词典相配合。

(2) 数据词典

这里我们暂不考虑三个加工的具体过程,只是对数据流图中的数据流和文件,共九个数据项给出数据词典的定义词条。

[1] 借书证=证号+单位+姓名+年龄+职务
+〔证章|密码〕

[2] 借书单=证号+姓名+1{书号+书名}5

[3] 索书单=借书单+可借标记

[4] 谢绝=〔非法证|不合格单|证单不符〕

[5] 已借出=索书单+已借出标记

[6] 供书单=证号+姓名+1{书号+书名+
〔可供标记|已借出标记〕}5

[7] 借书记录=借书单+还书日期

[8] 库存书目＝{书号＋书名＋作者＋出版社＋
出版年代＋库存总数＋借出册数}

[9] 借书登记卡＝{借书日期＋供书单}

需要对以上九个定义式中右端名字进一步给出第二层定义词条。

[1.1] 证号＝“0001”..“9999”

〔1.2〕 单位＝2{字母}24

[1.3] 姓名＝2{字母}24

〔1.4〕 年令＝“15”..“100”

[1.5] 职务＝〔“XUE”|“JIAO”|“ZHI”〕

[2.3] 书号＝1{字母}2＋“00001”..“99999”

[2.4] 书名＝1{字母}120

[3.2] 可供标记＝“KJ”

[4.1] 非法证＝“FEIFAZHENG”

[4.2] 不合格单＝“BUHEGEDAN”

[4.3] 证单不符＝“ZHENGDANBUFU”

[5.2] 已借出标记＝“YJ”

[6.5] 可供标记＝“KJ”

[7.2] 还书日期＝日期

[8.3] 作者＝姓名

[8.4] 出版社＝1{字母}120

[8.5] 出版年代＝“60”..“90”

[8.6] 库存总数＝“1”..“100”

[8.7] 借出册数＝“1”..“100”

需要给出的第三层定义词条是：

[1.2.1] 字母＝〔“A”..“Z”|“a”..“z”〕

[7.2.1] 日期＝“88”..“90”＋“/”＋“01”..“12”＋“/”＋
“01”..“31”

在第1个定义词条中的“证章”和“密码”为特殊的数据，这里未定

义。

2.9 判定表和判定树

在某些数据处理问题中，其数据流图的加工需要依赖于多个逻辑条件的取值，在这些取值的组合构成的多种情况下，执行不同的动作。这样一类问题最适于用判定表(Decision Table)作工具来表达。

让我们以一个简单的实例来说明什么是判定表。

在一本书的几页目录之后，读者发现一个表，名为“本书阅读指南”。表的内容给读者指明了在阅读过程中可能遇到的种种情况，以及作者针对这些情况给读者的建议(参看表 2.3)。表中列举了读者阅读时可能会遇到的三个问题，若回答是肯定的(判定取真值)，标以“Y”；若回答是否定的(判定取假值)，标以“N”。三个判定条件，其取值的组合共有八种情况。对读者的建议部分有四条建

表 2.3 读书指南判定表

		1	2	3	4	5	6	7	8
问题	你觉得疲倦吗?	Y	Y	Y	Y	N	N	N	N
	你对内容感兴趣吗?	Y	Y	N	N	Y	Y	N	N
	书中的内容使你糊涂吗?	Y	N	Y	N	Y	N	Y	N
建议	请回到本章开头重读	X				X			
	继续读下去		X				X		
	跳到下一章去读							X	X
	停止阅读，请休息			X	X				

议，并不需要每种情况都施行。这里把要施行的建议相应栏内标以“X”，其它建议相应的栏内留作空白。例如，表中的第三种情况，当读者已经疲劳，对内容又不感兴趣，还没读懂，这时作者建议去休

息。

早在程序设计发展的初期，判定表就已被当作编写程序的辅助工具使用了。由于它可以把复杂的逻辑关系和多种条件组合的情况表达得既明确又具体，因而给编写者、检查者和读者均带来很大方便。

判定表通常由四个部分组成，如图 2.15 所示，双线分割开的四部分是：

条件茬(Condition Stub)

动作茬(Action Stub)

条件项(Condition Entry)

动作项(Action Entry)

条件茬部分列出了各种可能的条件，除去某些问题中对条件的先后次序有特定要求外，通常判定表中条件茬所列的条件，其次序无关紧要。条件项是针对各种条件给出的多组取值，即多个条件所取真假值的组合。动作茬列出了可能采取的动作。这些动作的排列顺序并没有什么约束，但为便于阅读也可令其按适当的顺序排列。动作项是和条件项紧密相关的。它指出了在条件项的各组取值情况下应采取的动作。我们把任何一个条件组合的特定取值及其相应地要执行的动作(在判定表中贯穿条件项和动作项的一列)称为规则。显然，判定表中列出了多少个条件组合值，也就有多少条规则，即条件项和动作项有多少列。

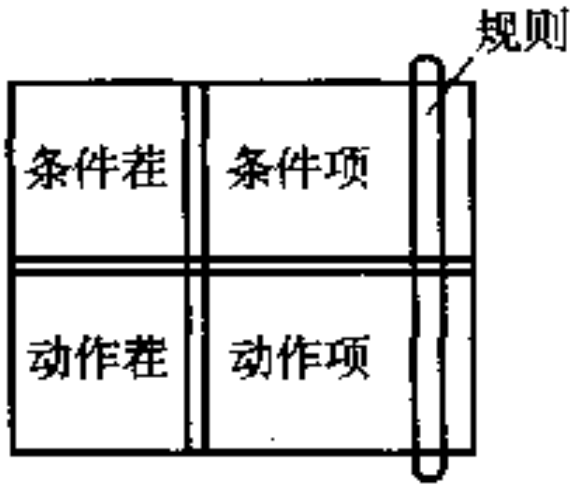


图 2.15　判定表的四个组成部分

在实际使用判定表时，常常先把它化简。若表中有两条或更多的规则具有相同的动作，并且其条件项之间存在着某种关系，我们便可设法将其合并。比如图 2.16 表示了两个规则的动作项一致，

条件项中的第三条件取值不同，这表明，在第一、二条件分别取真和假值时，第三条件取值任意，都要执行同一动作。也即要执行的动作与第三条件的取值无关。于是，我们便将这两个规则合并。合并后的第三条件用“—”表示与取值无关。

与此类似，无关条件项“—”在逻辑上又可包含其它的条件项取值，具有相同的动作的规则还可进一步合并。如图 2.17 所示。

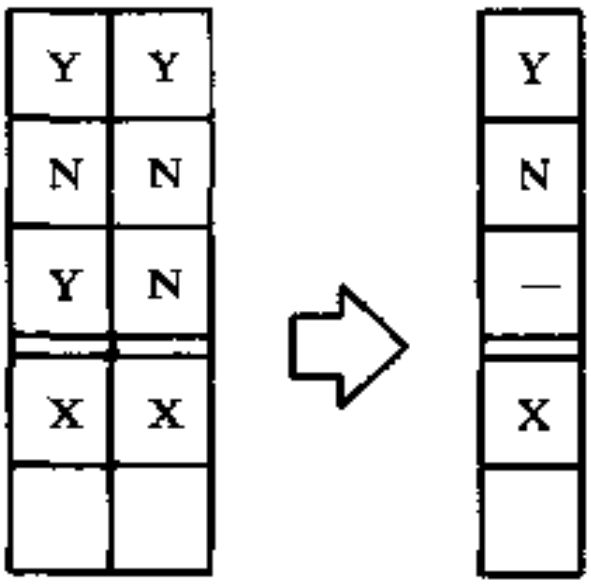


图 2.16　两条规则合并成一条

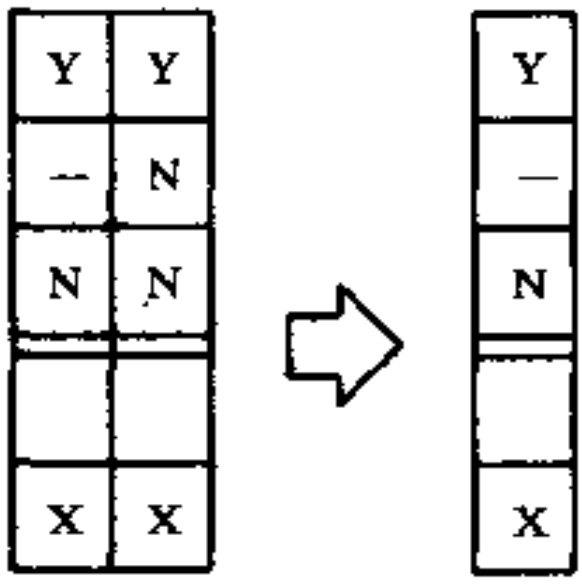


图 2.17　判定表规则进一步合并

按这一规则合并的办法，可把前述“读书指南”判定表加以简化，如表 2.4 所示。

以下结合实例给出构造判定表的方法。若问题要求：“……对功率大于 50 马力的机器、维修记录不全或已运行十年以上的机器，应给予优先的维修处理……”。这里假定“维修记录不全”和“优先维修处理”均已在另处有更严格的定义。按以下五步建立判定表。

（1）确定规则的个数。这里有三个条件，每个条件有两个取值，故应有 2×2×2＝8 种规则。

（2）列出所有的条件茬和动作茬。

（3）填入条件项。为防止遗漏可从最后一行条件项开始，逐行向上填满。如第三行为

YNYNYNYN

第二行为

YYNNYYNN

等等。这样得到的判定表如表 2.5 所示。

(4) 填入动作项。

(5) 化简。简化后的判定表见表 2.6。

判定表最突出的优点是，它能把复杂的问题按各种可能的情况逐一列举出来，简明而易于理解，也可避免遗漏。它的不足之处是不能表达重复执行的动作，例如循环结构。

		1	2	3	4
问题	你觉得疲倦吗？	—	—	Y	N
	你对内容感兴趣吗？	Y	Y	N	N
	书中内容使你糊涂吗？	Y	N	—	—
建议	请回到本章开头重读	X			
	继续读下去		X		
	跳到下一章去读				X
	停止阅读，请休息			X	

表 2.5 “机器维修”判定表

		1	2	3	4	5	6	7	8
条件	功率大于 50 马力吗？	Y	Y	Y	Y	N	N	N	N
	维修记录不全？	Y	Y	N	N	Y	Y	N	N
	运行超过 10 年？	Y	N	Y	N	Y	N	Y	N
动作	进行优先维修	×	×	×		×		×	
	作其它处理				×		×		×

表 2.6　简化的"机器维修"判定表

		1	2	3	4	5
条件	功率大于 50 马力吗?	Y	Y	Y	N	N
	维修记录不全?	Y	N	N	—	—
	运行超过 10 年?	—	Y	N	Y	N
动作	进行优先维修	×	×		×	
	作其它处理			×		×

判定树是判定表的变种，所有判定表能表达的问题均能用判定树来表达。事实上，判定树似乎比判定表更加直观。用判定树来描述具有多个条件的数据加工，更容易被用户接受。树的分枝表示各种不同的条件，随着分枝层次结构的扩展，各条件完成自身的取值。树枝的叶端处给出应完成的动作。以处理订购单的工作为例，画出的判定树在图 2.18 中给出。

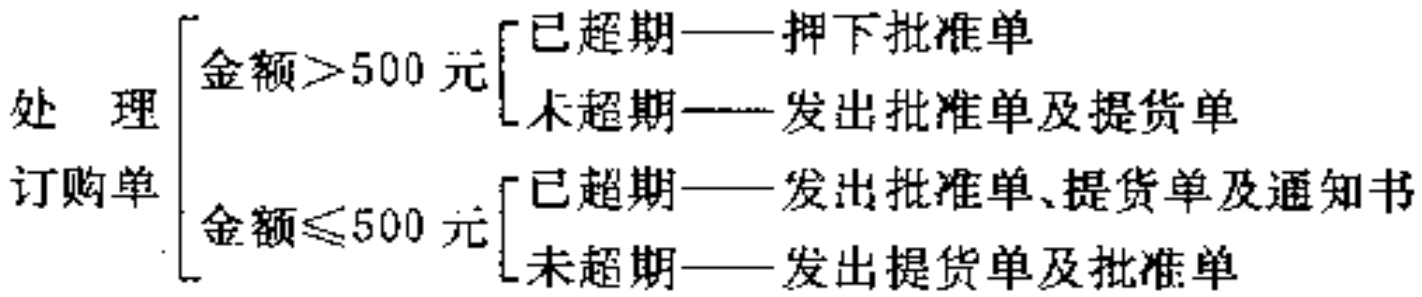


图 2.18　判定树实例——处理订购单

2.10　面向数据结构的分析方法

本章 2.2 节中提到问题的信息域中包括三方面内容，即信息流、信息内容和信息结构。所谓信息流、信息内容和信息结构也就是数据流、数据内容和数据结构。本章从 2.5 节至 2.9 节所介绍的都是从数据流出发展开的结构化分析方法。本节给出的面向数据

结构的分析方法则是侧重于从数据结构方面去分析和表达软件的需求。

以下讨论的两个面向数据结构的分析方法，其共同点是：

① 方法提供了一种手段，以帮助我们分析和表达需求分析工作最主要的目标—对象(object)和操作(或称加工、动作)。

② 假定数据结构是分层次的。

③ 方法要求，数据结构可由顺序的、选择的和重复的三种构造组合而成。

④ 提供一些步骤，使之可按此步骤把层次的数据结构映象为程序结构。

面向数据结构的分析方法与是面向数据结构设计方法的基础。

以下两节将介绍两个重要的面向数据结构的分析方法：结构化数据系统开发和 Jackon 系统开发。

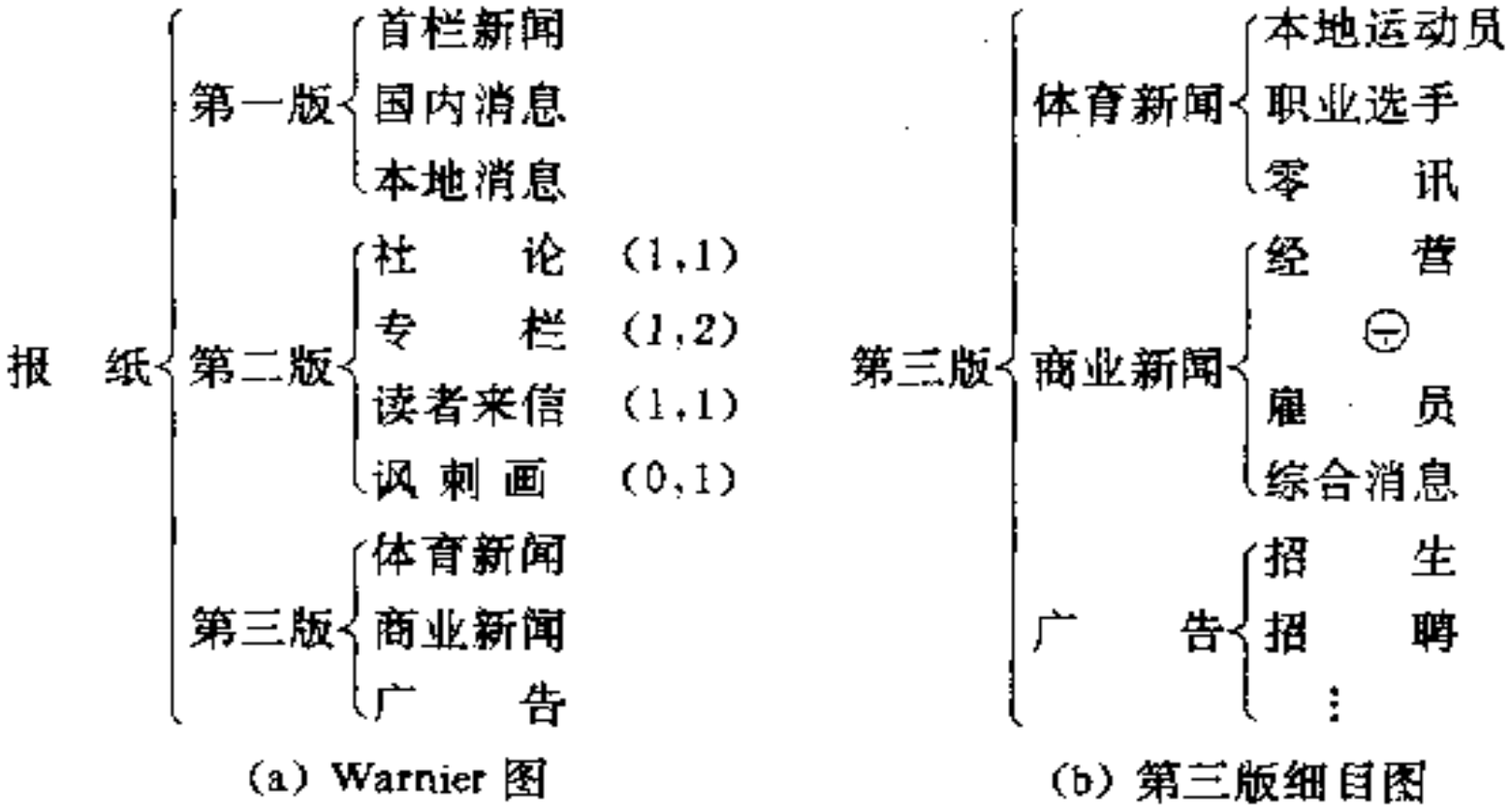
2.11 结构化数据系统开发

结构化数据系统开发(Data Structured Systems Development)方法也称 Warnier-Orr 方法，是由 J. D. Warnier 提出的。他利用三种构造，即顺序的、选择的和重复的构造来表示分层的信息，并进而导出软件的结构。Orr 将其扩充，形成了结构化数据系统的开发方法。该方法既考虑了信息流和功能特性，也考虑了数据的层次关系。

(1) Warnier 图

Warnier 是表示层次信息的一种紧凑而直观的方式，很容易被人们理解。这里以报纸的自动编辑系统为例，如果我们看到一个图 2.19(a)，就很容易想到，这必定是报纸各版的构成。而图 2.19(b)则是第三版的详细栏目。其实各栏目的结构关系一目了然，无需过多解释。花括号表示层次关系，括号内从上到下是顺序的信息项。

(a)图中第二版内容的右端圆括号内的数字有特定含意:注有(1,1)的表示社论和读者来信都占用一栏,注有(1,2)的专栏。应占用2栏,讽刺画所注(0,1)表示可有可无,若有则占1栏。在(b)图的商业新闻的内容中有符号⊕,这表示经营和雇员仅取其中之一。



(a) Warnier 图　　(b) 第三版细目图

图 2.19　报纸版式

(2) 结构化数据系统开发方法简称 DSSD 方法,它并不以考察信息的层次开始,而是首先研究"应用环境"。即从信息的产生者和接收者的观点,来观察数据如何在两者之间运动。然后,用类似 Warnier 的表示方法来表达问题的功能。最后,再用 Warnier 图给出问题的结果。使用这一方法作需求分析会涉及到信息域的所有属性:数据流、数据内容和数据结构。

以下结合邮电订购业务为例,说明这一方法的步骤。为表明邮电订购业务的情况,我们使用了数据流图。其实数据流图并不是 DSSD 的要求,只是因为我们比较熟悉它。

在图 2.20 中可看出,售货员接到订购货物的信件或电话后,作好记录,并建立一个订购文件。文件的内容包括:顾客姓名、地址、订购日期、货号、批号、品名、规格、数量、单价及总计。需要给一

个订购号，并把这一订购号传送给发货部。发货部即可利用订购文件中的信息。其它的业务活动如会计、管理等也可对其进行存取。

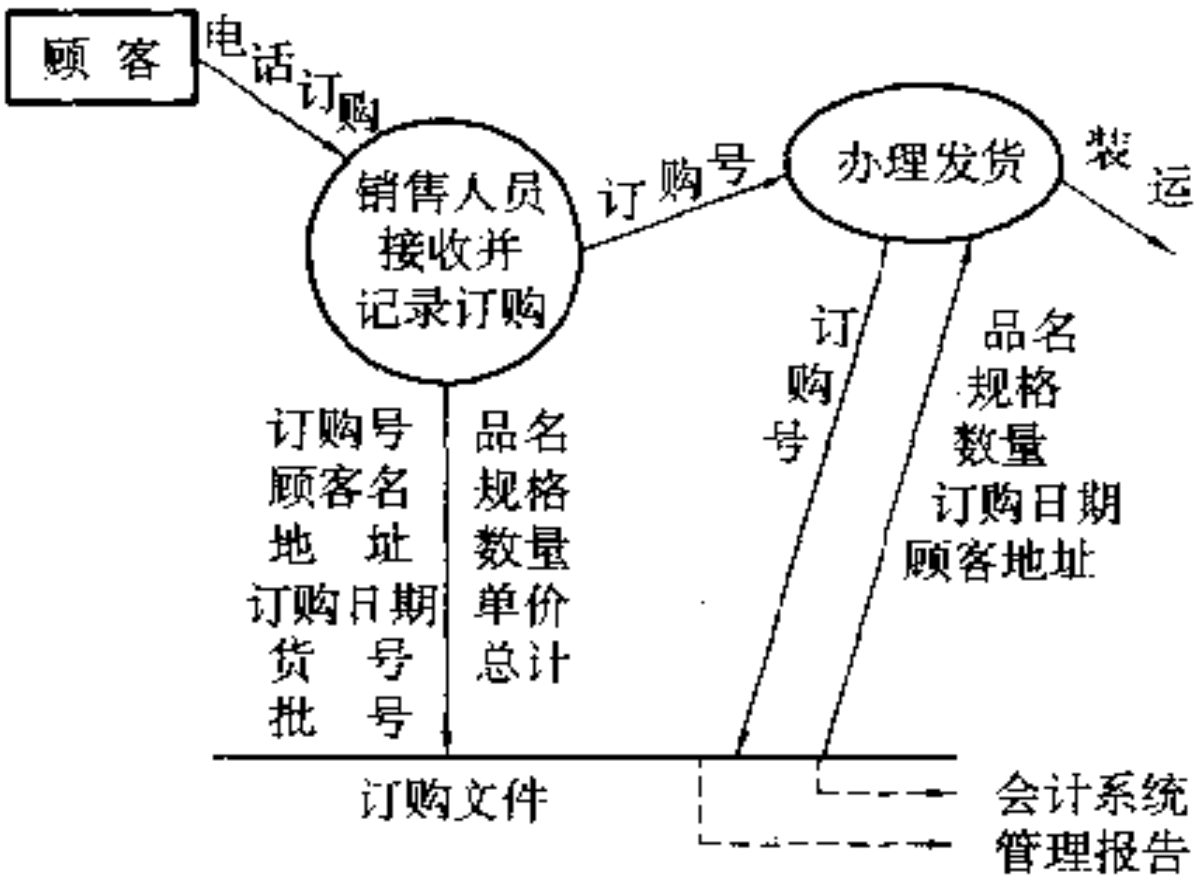


图 2.20 邮电购货业务信息流程

(3) 应用环境

为了确定问题的应用环境，对问题作出描述，需要回答以下三个问题：

① 要处理的信息项有哪些？

② 谁是信息的产生者和接收者？

③ 每个信息的产生者或接收者怎样看其它的订户？

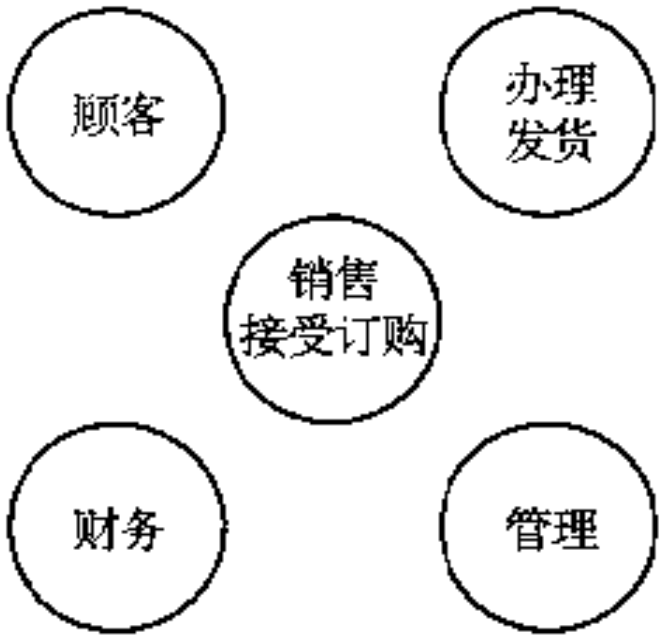
DSSD 以实体图为机制回答了这些问题。恰巧，实体图又非常像数据流图，但图中所用符号的含意却完全不同。

图 2.21(a)中给出了五个信息的产生者和接收者。(b)为销售实体图，表明了收订部的业务。这是从收订的观点把销售与顾客之间的接口全部表示了出来。(c)为应收帐目实体图，(d)为顾客实体图，(e)为顾客服务实体图。检查完各实体图的正确性以后，便可画出综合实体图(图 2.22)。图中的虚线内是订购系统业务，虚线称为问题的边界。如果我们进一步把界内的业务暂隐蔽起来，可以得

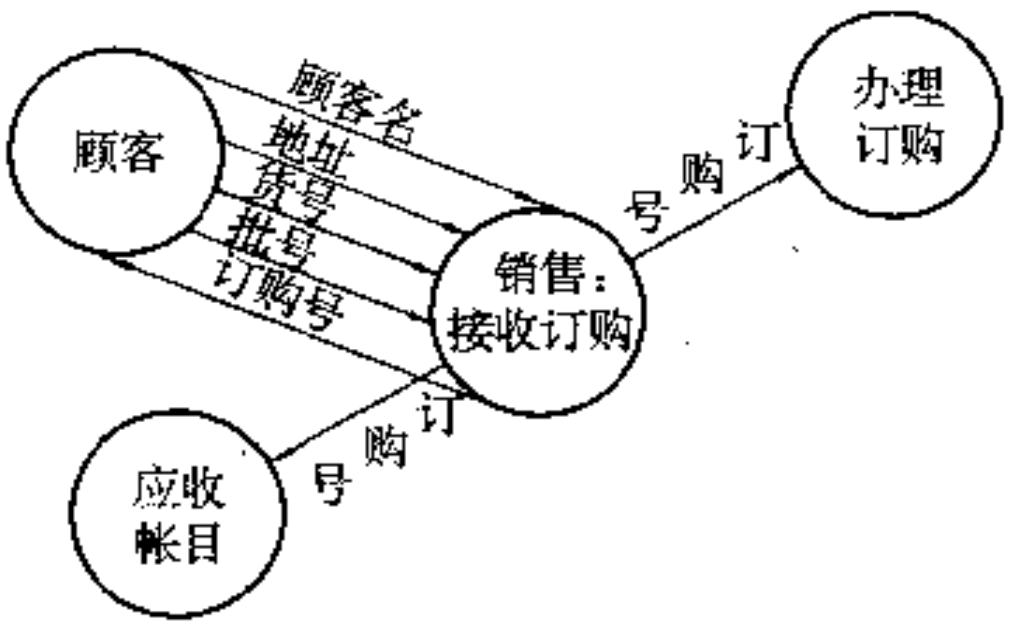
到图 2.23。显然，穿越边界的信息都要得到订购系统的处理。

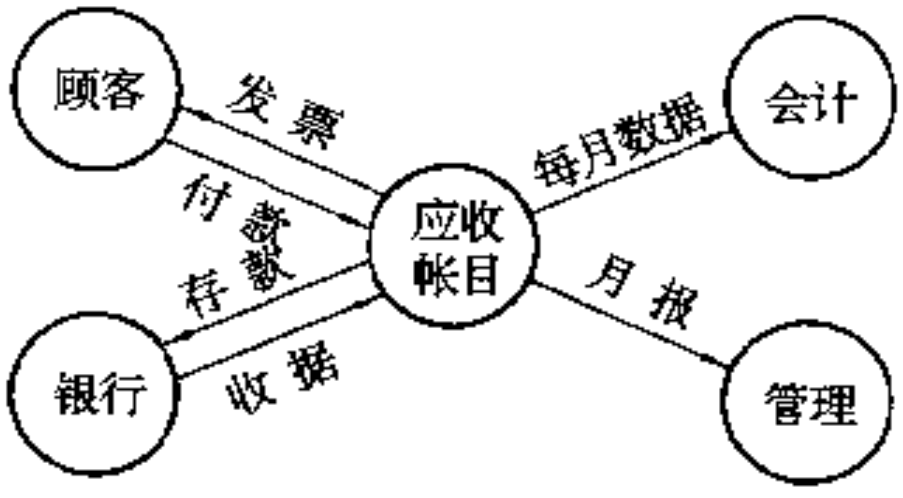
(4) 问题功能

研究跨越边界的信息流可以弄清楚自动订购系统应该实现的功能。图 2.23 对这些信息均给了编号。使用类似 Warnier 的表示法，我们可以把信息和加工联系起来，称之为作业线图(图 2.24)。从概念上讲，这个图起着数据流图的作用。注意，这个图是以图 2.23 中最大编号信息流开始直到最小编号信息流的顺序画出的。

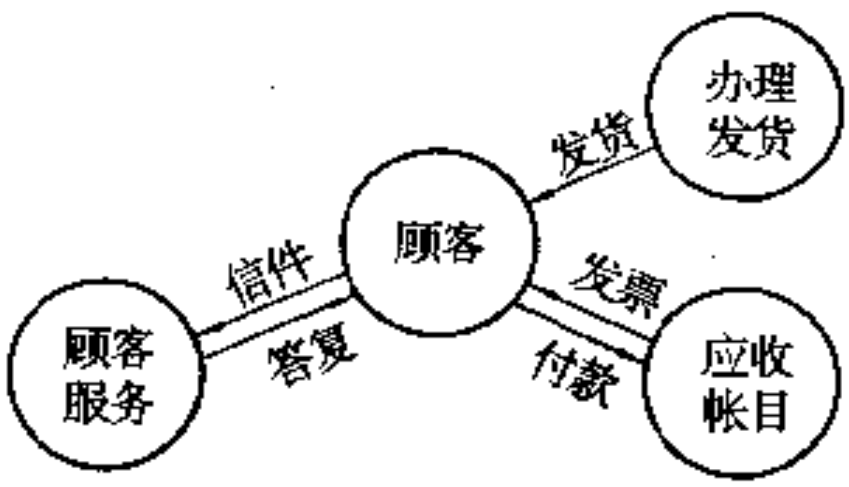


(a)

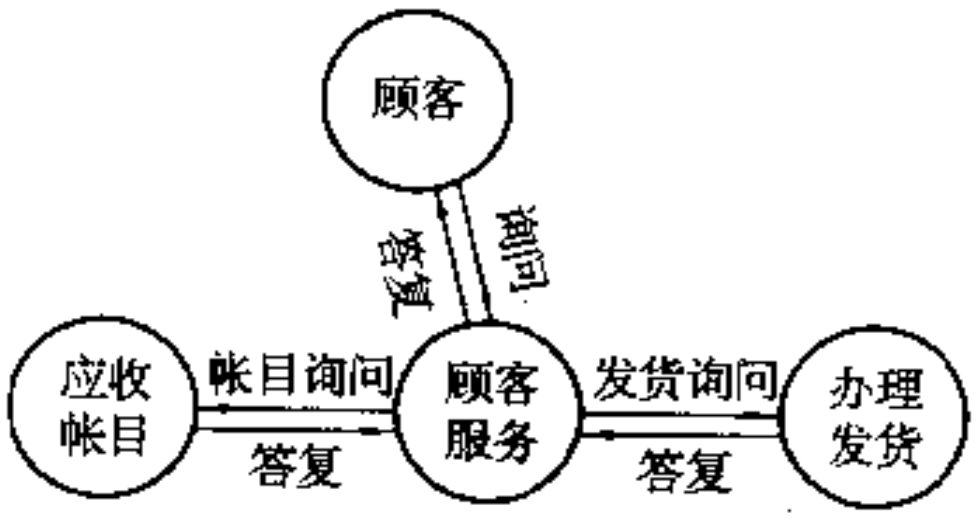



顾客
发票
付款
应收
帐目
每月数据
会计
存款
银行
收据
月报
管理

(c)


办理
发货
发货
顾客
信件
发票
顾客
服务
答复
付款
应收
帐目

(d)


顾客
答复
询问
应收
帐目
帐目询问
顾客
服务
发货询问
办理
发货
答复
答复

(e)

图 2.21

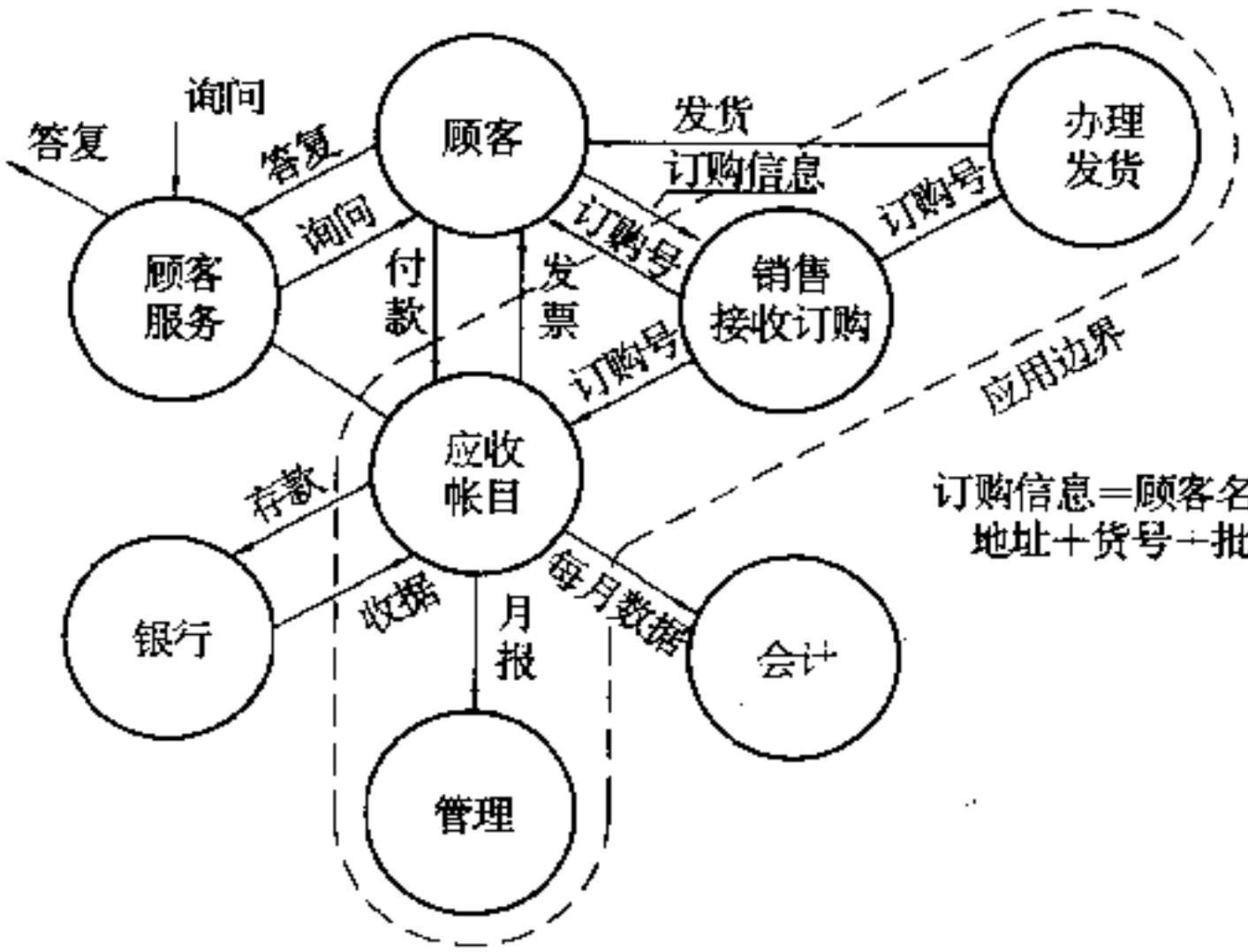


图 2.22　综合实体图

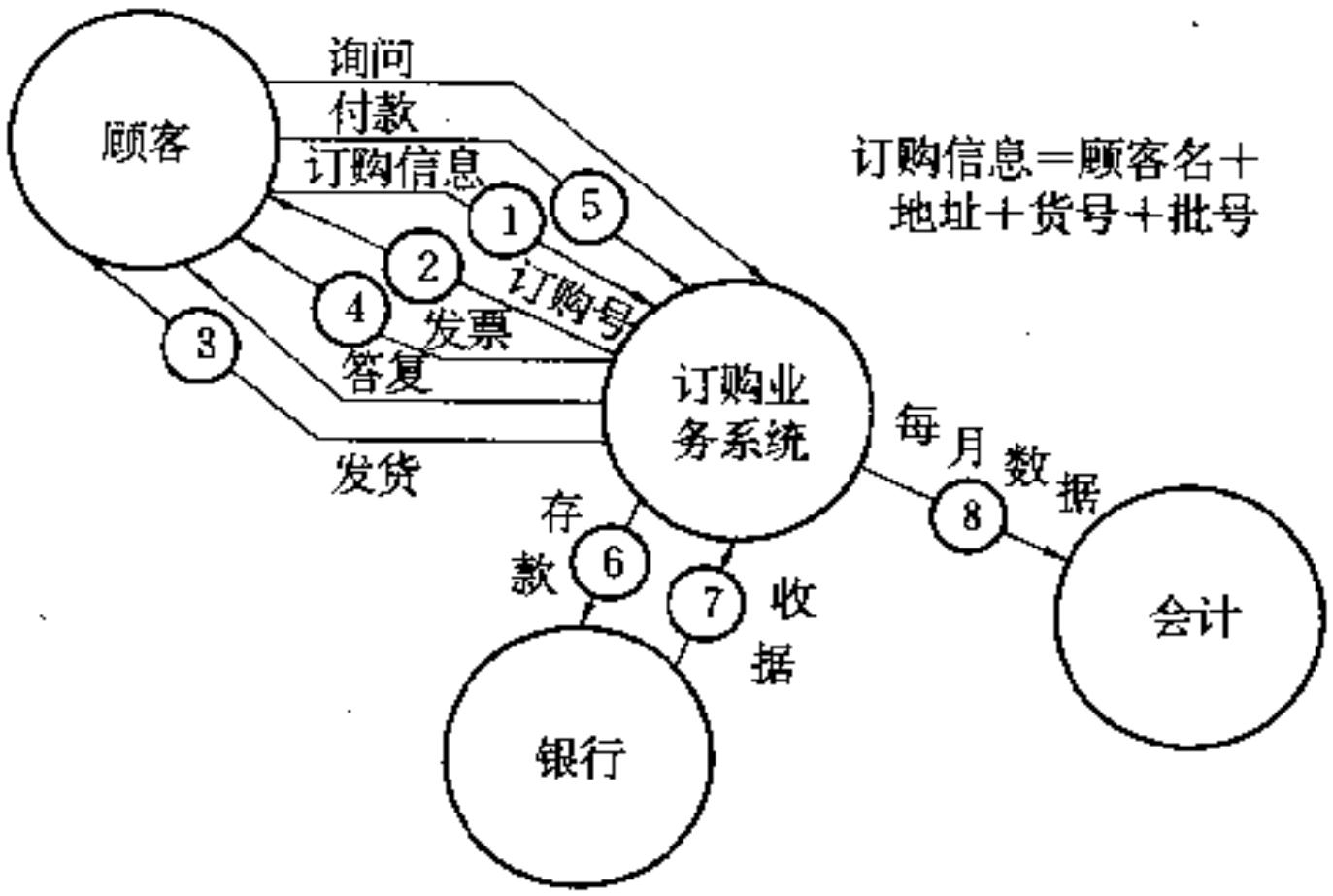


图 2.23　应用级实体图

每个信息流项是前面编号信息项与产生所需项的过程结合起来得到的。在图 2.24 中我们看到，自左向右，月报是银行接收信息及使用报告生成过程得到的。加号表示信息与加工联系在一起。接收信息则是由银行存款信息及其相关的过程得到的，该过程处理存款并给出收据。依此向右推进，直至最小号码的订购信息为止。在这个作业线图中，每个加工过程由一个处理说明细化，该说明包括输出、动作、动作的频率及输入。下一节将再用 Warnier 图来表示每一加工的细节。

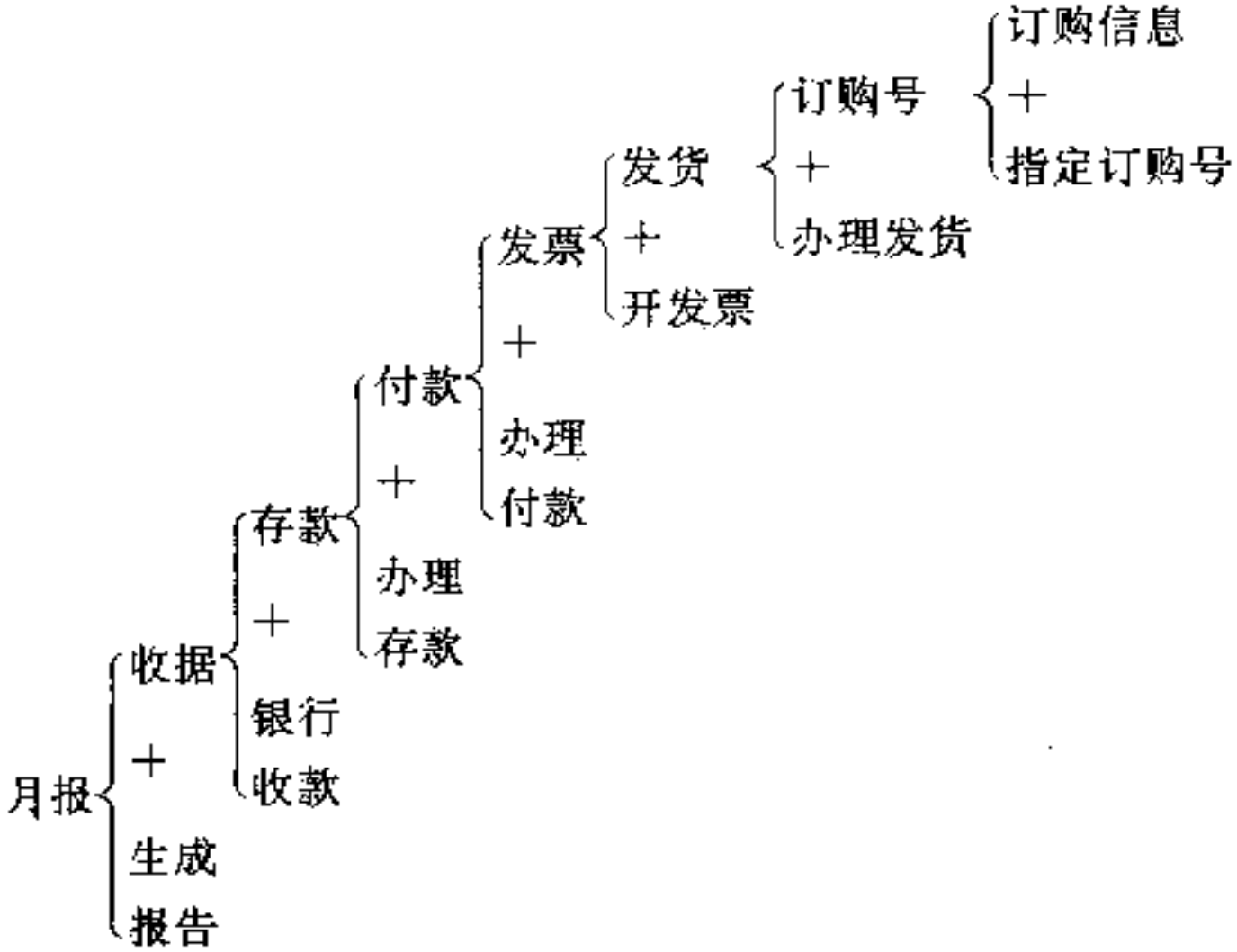


图 2.24　作业线图

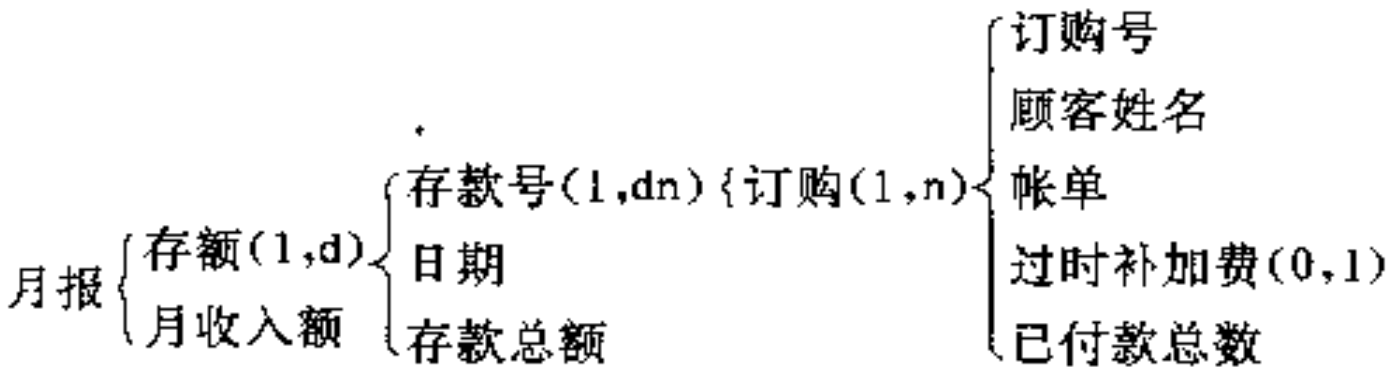
(5) 问题的结果

DSSD 要求对系统的输出建立雏型，以表明主要的系统输出及构成输出的信息项组织。有了这一雏型就可利用 Warnier-Orr 图来模拟信息的层次结构了。其实，Warnier-Orr 图和 Warnier 图的差别是很小的。图 2.25(a)和(b)是自动订购系统要求输出的月报雏型和它所对应的 Warnier-Orr 图。

月　　报

存　款		发　票　信　息				
存款号 日期		订购号	顾客名	帐单	过时补加费	已存款总数
——— ——		———	———	——	———	———
		———	———	——	———	———
		———	———	——	———	———
存款总额						———
——— ——		———	———	——	———	———
		———	———	——	———	———
存款总额						———
月收入额						———

(a) 月报格式



(b)

图 2.25　月报的 Warnier-Orr 图

2.12　Jackson 系统开发

Jackson 系统开发(Jackson System Development 简称 JSD)与 Warnier 方法及 SDDS 方法很相似,它着重于研究现实世界信息域的模型。使用这一方法按下列步骤进行:

① 实体动作分析——设法弄清系统要产生或使用的实体及对实体施加的动作。

② 实体结构分析——对实体的动作按时间发生前后排序,用

树状的 Jackson 图表示。

③ 定义初始模型——实体和动作被当成一个过程模型来表示，确定模型和现实世界的联系。

④ 功能描述——描述对应于所定义动作的功能。

⑤ 决定系统时间特性——估计并描述过程的时间特性。

⑥ 实现——设计硬件和软件。

以上的后三步与系统和软件的设计相关。这里仅讨论前三步。

(1) 实体动作分析

分析实体动作首先用简明的自然语言描述问题。以下结合一个实例进行分析，该例是学校交通服务系统 USS，概述如下：

“某学校的两个校园相距一公里多，为使学生在两校园之间的穿行不致影响按时上课，学校计划建一专用交通服务设施。该设施仅包括一个沿轨道运行的车辆，往返于两校园的两个车站之间。每站设有一呼叫按钮，供需搭乘运载车的学生使用。当学生来到车站时，可按下呼叫按钮。若车本来就停于此站，则可立刻搭乘，驶向对方车站。若车正在运行中，呼叫者需等待，至车到达对方站，让站上的学生(如果有)搭上，再开出。若呼叫时，该车正停在对方车站，得到信号后，闸讯开来，以供搭乘。在未得到呼叫信号的情况下，无人乘车，通常该车停在某个车站上。”

以下从这段对问题的描述中选定实体和动作。可能作为实体的名词有：学校、校园、学生、上课、车辆、车站、按钮。其实，我们并不关心“学校”、“校园”、“上课”、“学生”和“车站”。这些都处在要解决问题模型的边界之外，它们不可能作为实体。去掉这些之后，只考虑“车辆”和“按钮”。

动作是在特定时刻对实体实施的，可从以上对问题描述的动词中选择。可供考虑的动作有穿行、到达、按、搭乘，开出和等待。由于“穿行”一词涉及到学生(学生并未被选作实体)，又由于“等待”表示一种状态而不是动作，我们去掉这两词，只取“到达”、“按”和

“开出”。

注意，当我们选实体和动作时，实际上已把要开发系统的范围划定了。例如，去掉“学生”后，便把生成“今天有多少学生要乘车”的信息等问题排除了。不过，实体和动作的范围随着分析工作的进展还可能有些变动。

(2) 实体结构分析

在 JSD 方法中，实体结构表达了实体随时间变化的动作。图 2.26 中给出了对实体的三种典型动作，即顺序的、选择的和重复的。在实体结构图中，顺序型是指在实体下的动作按时间先左后右；选择型使用了符号“。”，表示只取其中之一；重复型表示了对标有“*”的动作重复执行多次。

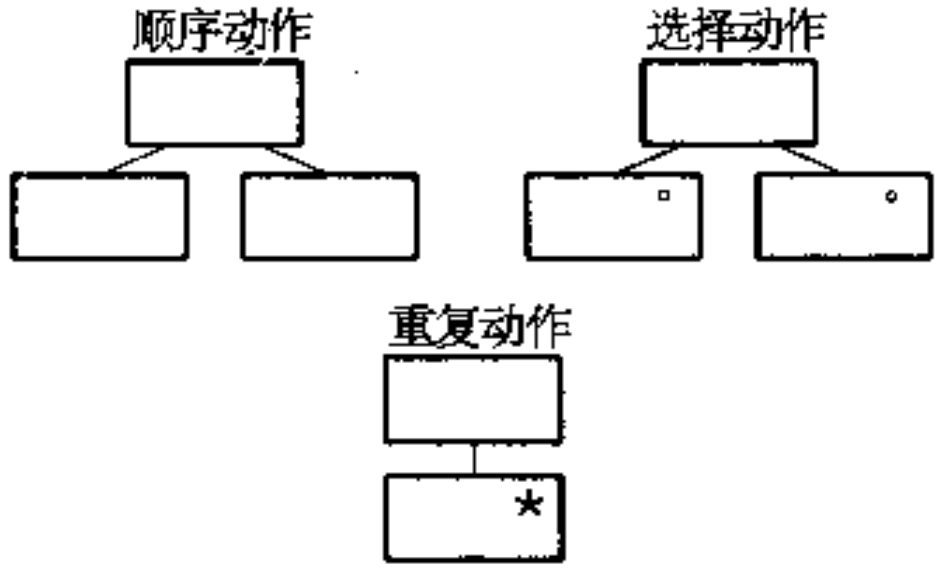


图 2.26　结构图表示法

图 2.27 的(a)和(b)分别表示了实体车辆和按钮的结构图。在(a)图中，我们可以看出，车辆的活动的开始和结束都是在 1 号站。作用于实体的动作只有到达和开出(均在树叶上)。车辆从 1 号站开出，然后往返于两站之间，最后返回 1 号站。我们发现，通常到达某站后，紧跟着必定是从该站开出。表示这一现象为

arrive(i)

和　leave(i)

其中 i 为站号。

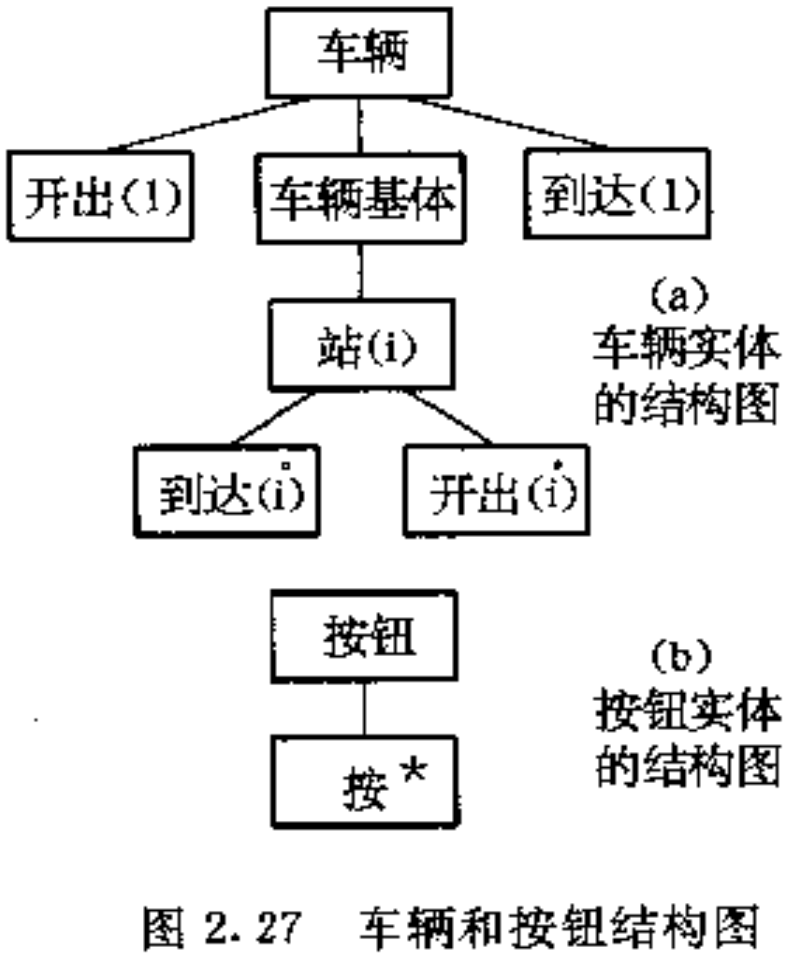


图 2.27　车辆和按钮结构图

对于那些无法用 JSD 记法表达的限制，常常需要对结构图加上注解。比如："i 的值只能取'1'或'2'，两次连续出现车站时，i 的值必定改变。"

图 2.27(b)是施加于实体按钮的重复动作——按。

从这个实例可看出结构图是对实体实施的或由实体实施的动作规格说明的时序表示。正是由于这一点，它对现实世界的描述比起简单的实体和动作表来，要准确得多。不过对每个实体构造结构图时，还可能需要伴有文字注解。

(3) 定义初始模型

以上两步只是现实世界的一种抽象描述，完成的工作是把实体和动作选定，并用结构图来表达它们之间的关系。这一步则要对系统构造规格说明，使其成为现实世界的模型。系统规格说明图所使用的符号见图 2.28。在某个过程传送信息流(如写记录)，另一过程接收该信息流(如读记录)时，便出现一个数据流连接(data stream connection)。箭头表示信息流方向，圆圈表示要被放入一个容量无限的先进先出(FIFO)缓冲区的数据流。在一个过程直接检验另一过程的状态向量时，会出现状态向量连接(state vector connection)。此时箭头仍表示信息流方向，而菱形表

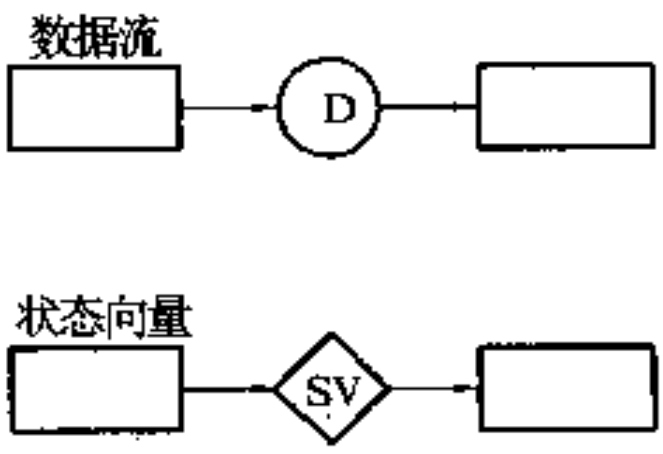


图 2.28　系统规格说明图(SSD)所用符号

示状态向量。在检查一些机电设备状态的过程控制问题时，这种联系是很常见的。图中矩形框中带有“0”后缀者表示的是现实世界过程，带有“1”后缀者表示的是系统模型过程。

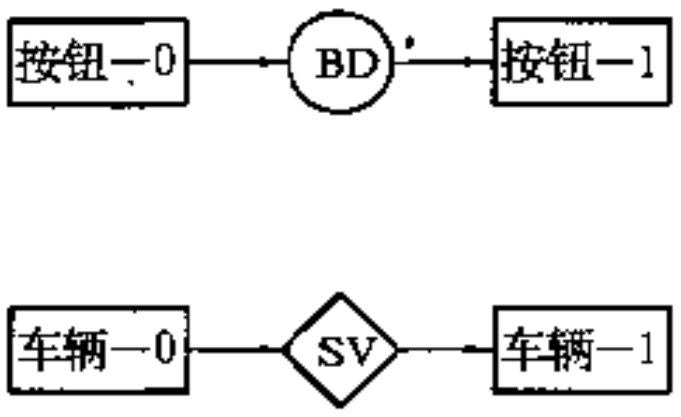


图 2.29 USS 系统的系统规格说明图

图 2.29 给出了实例 USS 的系统规格说明图。只要有可能，我们希望使用数据流来联系模型过程和现实世界实体，因为在模型的特性和现实世界之间有着直接的对应关系。在我们的实例中，当按下按钮时，便发出一个脉冲。这个脉冲可作为一个数据流连接传送给“按钮—1”的过程。不过我们假定，监视车辆到达或离开的传感器并不发出脉冲，除非我们确定合上了电器开关。开关的状态(合上或打开)可以被存取。因而，需要一个状态向量连接。

模型过程的内部细节可使用结构正文(structure text)来描述。结构正文与结构图(图 2.26)给出了相同的信息，同样包括顺序、选择和重复三类，只是它以正文格式表示。“按钮—1”的结构正文是：

```
BUTTON—1
  read BD;
  PUSH-BDY itr while BD
    PUSH;
    read BD;
  PUSH-BDY end
BUTTON—1 end
```

BUTTON—1 的结构与 BUTTON—0 的结构完全一样，只不过在“按钮—0”的结构正文中，读(read)操作把现实世界连接到系统中。

如前所述，过程 Shuttle－1 不能利用数据流连接与现实世界的对应物联系。我们只好询问站上由于车辆到达或离开而合上或打开的开关。系统过程必须不断地检查现实世界的实体，以保证所有动作无遗漏地获知。这一点利用操作 getsv（意为获取状态向量）来实现，它可取得现实世界实体的状态向量。很可能系统过程在状态向量改变以前多次获取到状态向量的值，而模型过程又被设计成能够反映状态向量的“途中”值。车辆-1 的结构正文描述为：

```
SHUTTLE－1 seq
    getsv SV;
WAIT-BDY itr while WAIT1
    getsv SV;
WAIT-BDY end
LEAVE(1);
TRANSIT-BDY1 itr while TRANSIT1
    getsv SV;
TRANSIT-BDY1 end
SHUTTLE-BDY1 itr
    STATION seq
        ARRIVE(i);
        WAIT-BDY itr while WAITi
            getsv SV;
        WAIT-BDY end
        LEAVE(i);
        TRANSIT-BDY itr while TRANSITj
            getsv SV;
        TRANSIT BDY end
    STATION end
SHUTTLE-BDY end
```

ARRIVE(1);

SHUTTLE-1 end

状态值 WAIT 和 TRANSIT 表示“到达开关”和“离开开关”的值。现实世界过程 SHUTTLE-0 对开关的状态作了翻转。系统过程 SHUTTLE-1 执行操作 getsv 获知到这一状态变化。图 2.30 把 SHUTTLE-1 的结构正文表达为结构图。

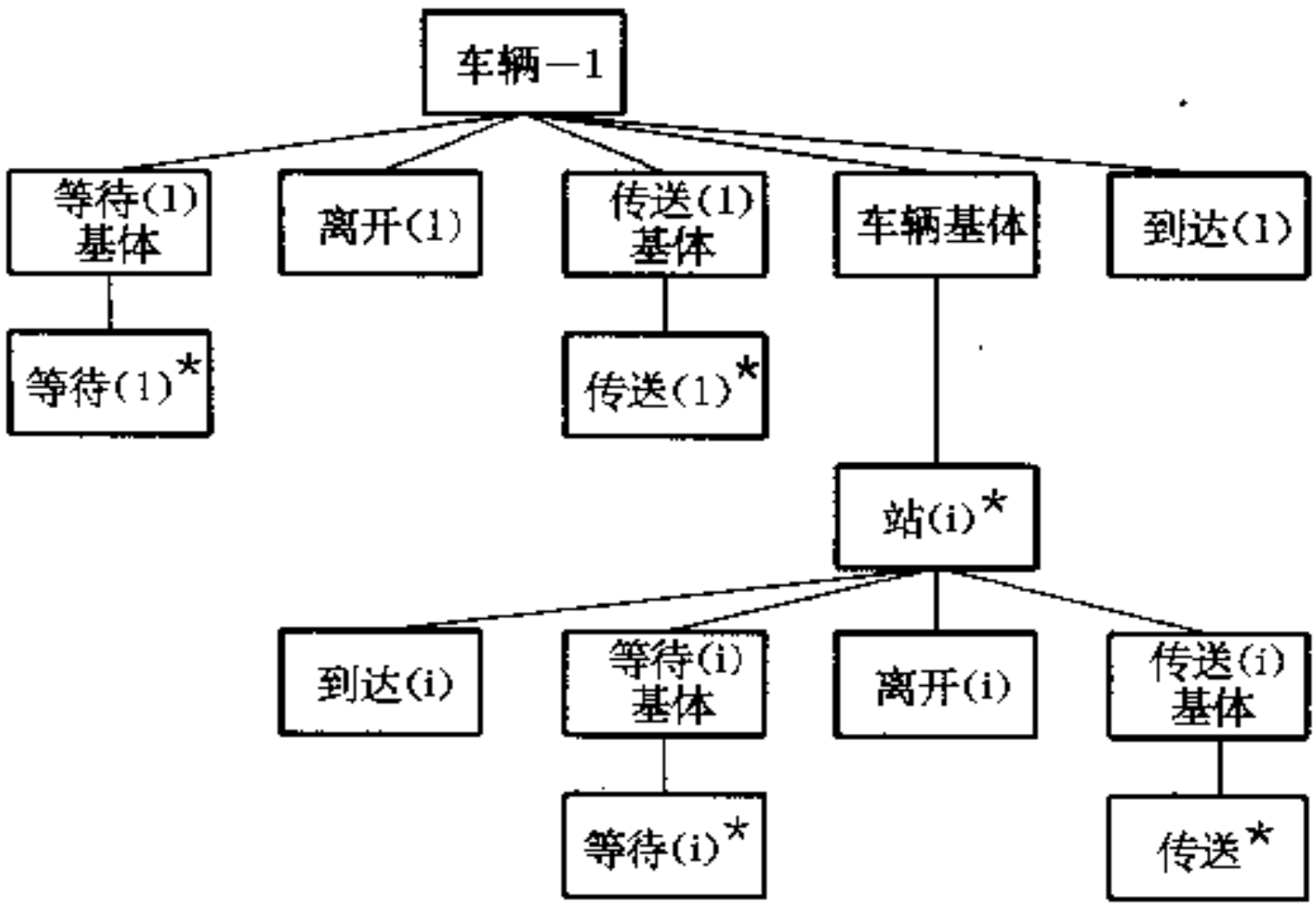


图 2.30　对应于结构正文的结构图

JSD 的下面步骤，可自然过渡到软件设计阶段。这里暂不讨论。

2.13　结构化分析与设计方法

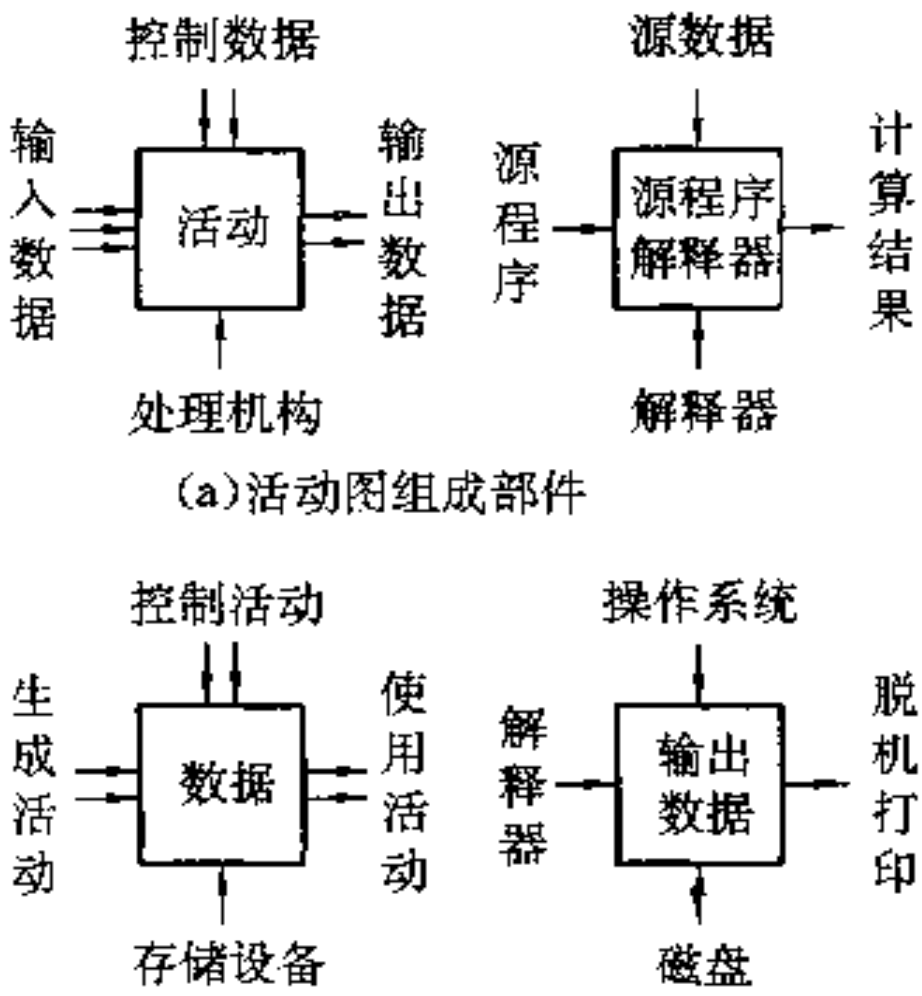
结构化分析与设计方法(SADT)是 D·T·Ross 等人在 1977 年提出的。这一方法把图形语言与使用语言的方法和管理原则结合起来。SADT 语言被称为结构分析语言(Language of Structured

Analysis)或SA 语言。使用这种语言表达需求分析所画的图类似于土建和机械工程中使用的工作蓝图。但应注意,SADT 和本章 2.5 节介绍的结构化分析方法没有直接联系,不要混为一谈。

SADT 模型是由一组有序的 SA 图构造而成的。每个 SADT 图单独画在一页纸上,它是一张格式固定的工程图纸。图中通常包含有 3 个至 6 个结点以及连接结点的弧。SA 图的两个基本类型是:活动图(Activity Diagram—actigram)和数据图(Data Diagram—datagram)。活动图的结点表示活动,弧表示活动之间的数据流。因此,活动图是数据流图(DFD)的另一形式,但要注意,不可和数据流图混淆。数据图表示结点内数据对象及弧上的活动。因而,数据图和活动图是对偶的。在实践中,活动图应用得更为广泛。不过数据图由于以下两个特点,也是很有用的:①它能指明被一个给定数据对象所影响的所有活动;②可利用由活动图构造数据图的办法去检查某个 SADT 模型的完全性和一致性。

图 2.31 给出了活动图和数据图结点的形式。请注意,每个结点上有四种不同类型的弧。结点左边进来的弧是输入,结点右边引出的是输出,顶端进入的弧是控制,底部进入的是机构。在 SA 图中,每一结点四个弧规定了结点之间的联系。一个结点的输出可作为输入和控制提供另一结点,某些结点的输出也可成为整个系统对外部环境的输出。每一结点的输出必须联到其它结点或是外部环境,它的输入和控制也必须来自其它结点的输出或是来自外部环境。一张图的输入、控制和输出必须联到另一页去。这种跨页的联线应以编号标出,防止搞乱。

在活动图中,输入和输出是数据流,机构是处理器(机械的或是人)。控制是活动所需数据,不被修改。在数据图中,输入是生成数据对象的活动,输出是使用数据对象的活动,控制是对结点活动条件的一种约束。这两种图的控制都是由外部环境和其它结点的输出所提供的。图 2.31 右面的部分是活动图和数据图结点示例。



(a)活动图组成部件

(b)数据图的组成部件

图 2.31　活动图和数据图的组成

图 2.32 表明了 SA 图的结构特性。其中，I_1，是外部输入，C_1 是外部控制。A_{11}的输出是 A_{12}、A_{13}和 A_{14}的输入，A_{12}的输出是 A_{13}的控制。A_{12}和 A_{14}可作并行处理。A_{13}则需等待 A_{12}的输出。A_{13}的输出又反馈控制 A_{11}。最后得到输出 O_1 和 O_2。整个图包含活动 A_{11}、A_{12}、A_{13}和 A_{14}是活动 A_1 的扩展(在图的左下角)。由于 SA 图中每一个结点都可以像图 2.32 那样扩展，因而可以建立层次的 SA 图。

图 2.33 给出 SA 图的应用实例，这个活动图描述了软件需求分析中的活动情况。

总之，SADT 以一种清晰、精确的方式提供了理解和表达复杂需求的方法。它的主要特点是，自顶向下把高层的结点分解成为若干个下属图，对每个结点分清它们的输入、输出、控制和机构，活动图和数据图的对偶性以及为开发、审查和调整 SADT 模型所需要的管理技术。SADT 也可应用到其它类型的系统中，并不只限于软

件开发。它适合于大型、复杂的项目分析。

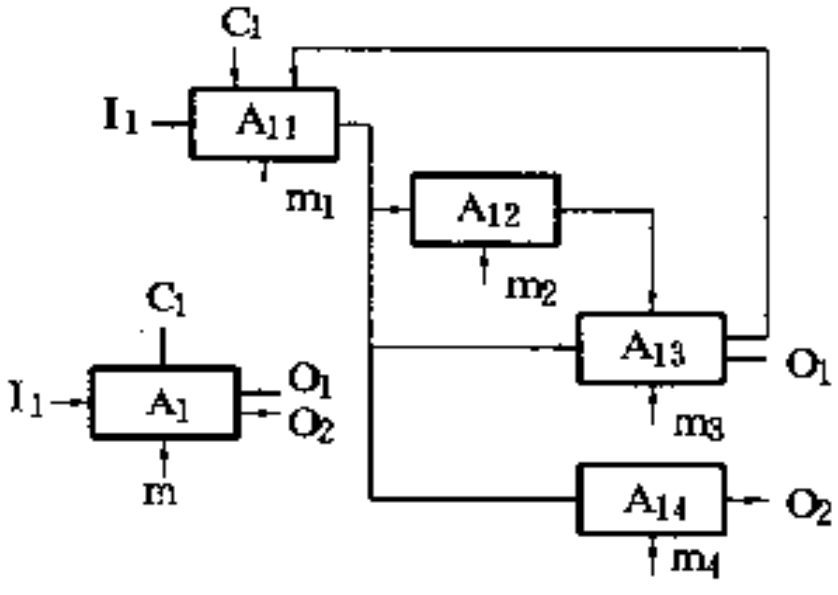


图 2.32　活动 A1 的扩展

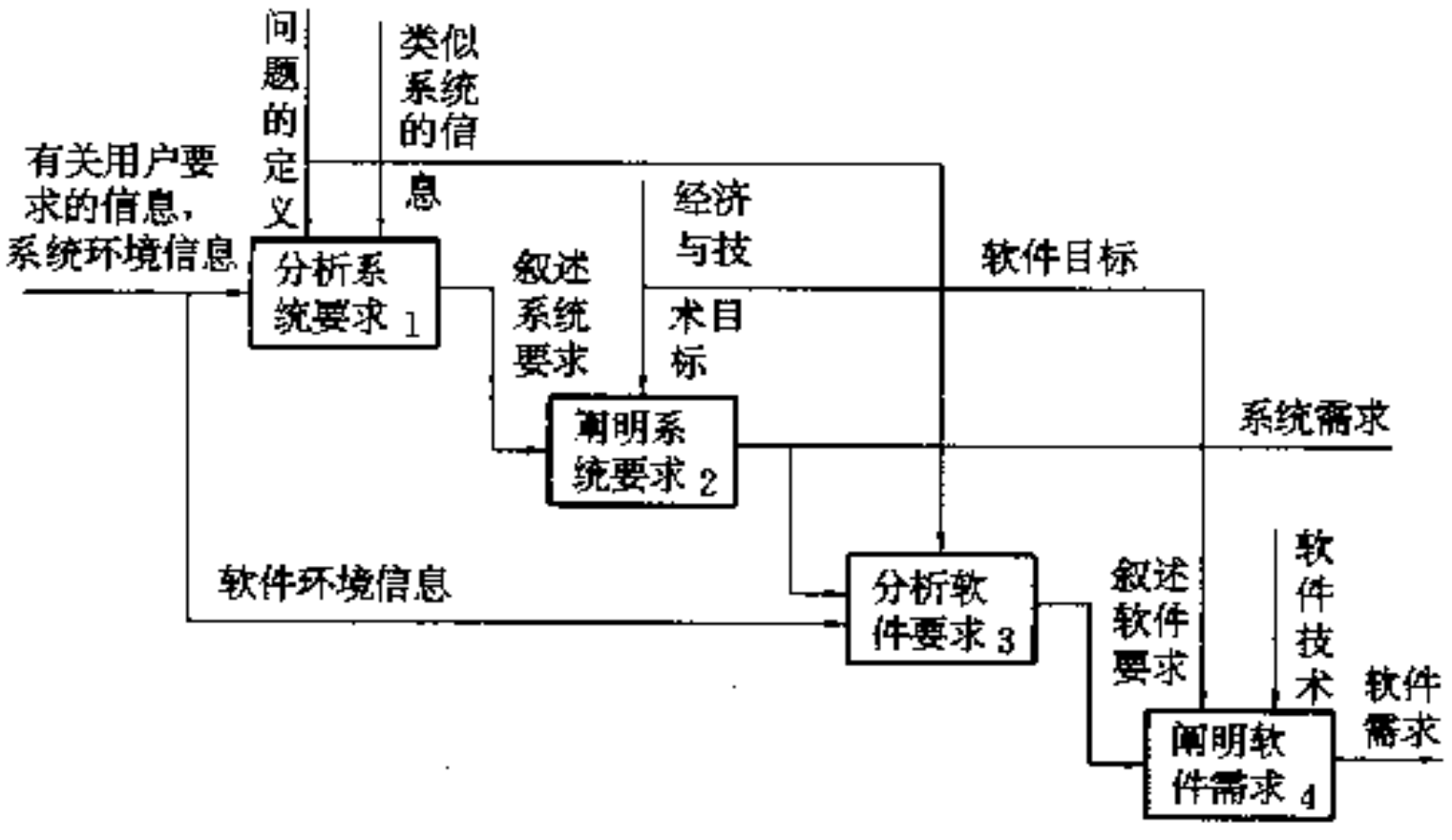


图 2.33　描述需求分析活动的活动图

第三章　软件设计

工程设计的概念在许多工程学科中早已为人们所熟悉。任何工程项目在施工以前，总要完成设计，这一点不会有什么疑问。然而，软件开发项目是不是需要设计、软件设计的含意是什么，这些问题却是随着软件工程学科的形成和发展，很晚才逐渐被人们所认识。在计算机学科领域里，除了计算机硬件有设计工作以外，多年来人们对软件工作的习惯说法是“程序设计”或者说“编写程序”。至今仍有不少人以为开发软件就是用某种语言编写出程序来。其实，这种说法并不全错，有它正确的成分，但若真的这样去理解软件开发工作，那便有极大的片面性。在第二章中，我们曾经把软件需求分析的必要性比作明确目标，没有明确需求的软件开发，必将是无的放矢。然而，在需求分析以后，进入了开发阶段，在程序编写的前后，像软件设计和软件测试这样十分重要的工作必须仔细考虑。

本章将围绕结构化设计方法讨论概要设计的有关问题。其它设计方法只作一简要介绍。

3.1　软件设计阶段的任务

软件开发工作经过需求分析阶段，完全弄清了需求，较好地解决了要让所开发的软件“做什么”的问题，并且已在软件规格说明书中详尽和充分地阐明了这些需求。进入设计阶段，开始着手对软件需求的实施，也即着手解决“怎么做”的问题。严格地说，软件产品的一些外部特性设计，如软件的功能、一部分性能以及用户使用特性等在需求分析阶段就已开始。这些问题的解决，实际上多少带

有一些“怎么做”的性质，因此有人称这些为软件的外部设计。

在软件设计阶段主要解决的是软件的总体结构和一些处理的细节。通常把设计阶段的工作分为两步：概要设计（也称总体设计或结构设计）和详细设计（也称过程设计）。

在概要设计阶段应完成的工作有：

① 程序结构的总体设计——决定软件的总体结构，包括整个软件系统分为哪些部分，各部分之间有什么联系以及已确定的需求对这些组成部分如何分配等方面。

② 数据结构设计——决定文件系统的结构或数据库的模式、子模式以及数据完整性、安全性设计。

③ 完成用户手册——对需求分析阶段编写的初步用户手册进行重新审订，在概要设计的基础上确定用户使用的要求。

④ 制定初步的测试计划——完成概要设计以后，应对测试的策略、方法和步骤等提出明确的要求。尽管这个测试计划是初步的，尚不十分完善，但在此基础上，经过进一步完善和补充，可作为测试工作的重要依据。

⑤ 概要设计评审——在以上几项工作完成以后，组织对概要设计工作质量的评审。特别着重以下几个方面：软件的整体结构和各子系统结构、各部分之间的联系、软件的结构如何保证需求的实现、确认用户接口等。

详细设计要完成的工作是：

① 确定软件各个组成部分的算法以及各部分的内部数据组织。

② 选定某种表达方法来描述各个算法。

③ 进行详细设计的评审。

软件设计的最终目标是要取得最佳设计方案。所谓“最佳”是指，在若干个候选方案中就节省研制费用、降低资源消耗、缩短研制时间的条件下，赢得较高的工作效率，以及较高的软件可靠性和

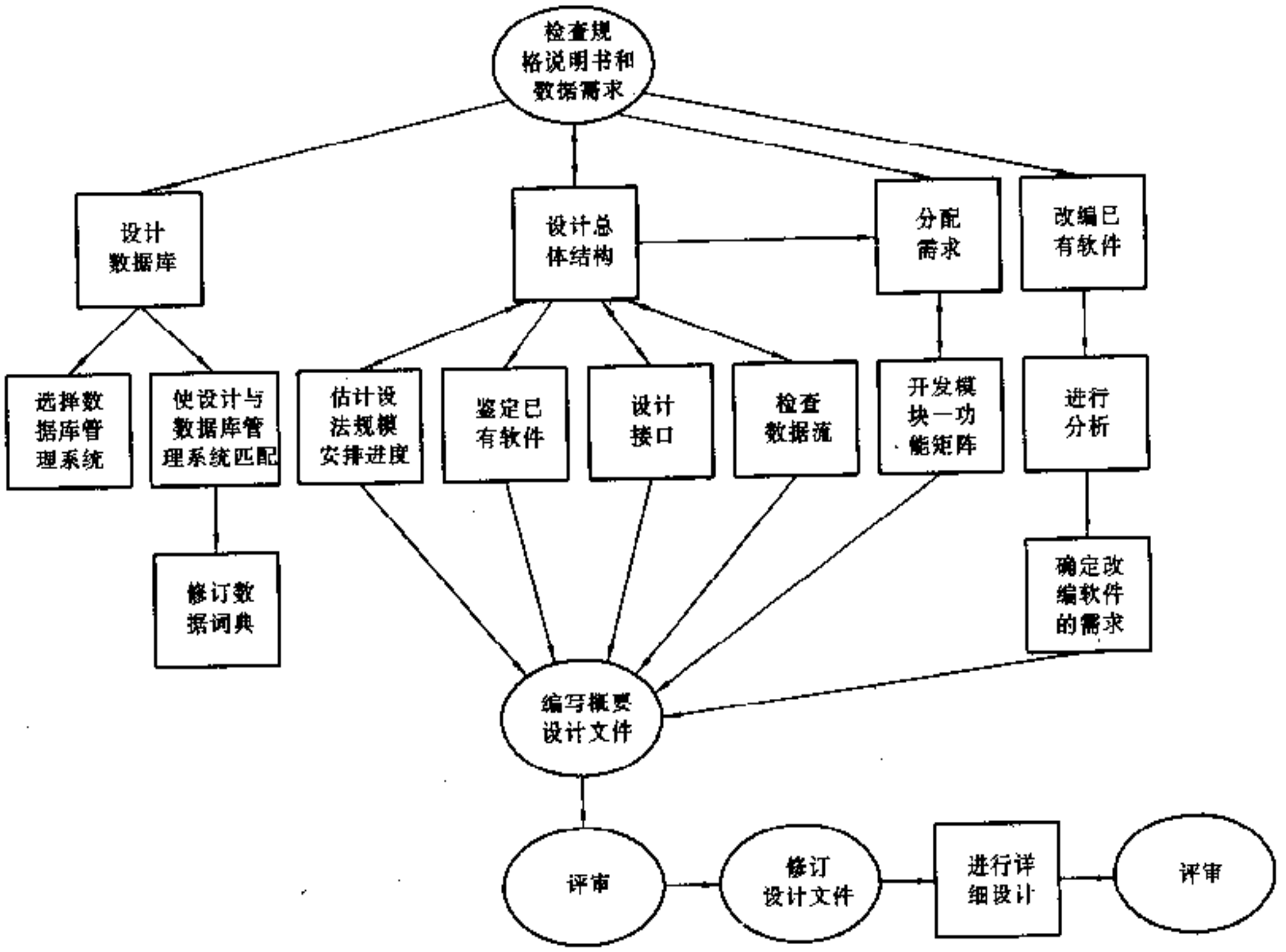


图 3.1　软件设计工作流程

可维护性。概要设计和详细设计之后的评审，都是要及时地发现和及时地解决软件设计中出现的问题，使之不致于把问题带到开发的后期阶段，造成后患。在评审以后，需要针对评审中发现的问题，对设计工作的成果进行必要的修改。

软件设计是软件开发的关键。在设计工作中要求软件设计人员付出创造性的劳动，又因为软件设计比代码编写重要得多。所以，通常人们十分重视设计工作的组织，一般选派有经验的软件人员参加，并注意选用适当的设计方法和设计表达方式。图 3.1 示出软件设计流程。

3.2 程序结构与结构图

如同上节所述，概要设计需要完成程序结构的总体设计，最主要的任务是解决如何把整个系统划分成若干个部分的问题。在软件开发项目的实践中，常常将各个部分继续划分，直至最小的基层单位，称为程序模块。实际上，每个程序模块就是将要实现某种特定功能的程序段。各个模块按某一形式组织在一起，称为程序结构。本节将围绕程序结构展开讨论。

(1) 树状结构与网状结构：

模块之间连接成的程序结构最普通的形式是树状的和网状的。

树状结构中，根部有一个顶层模块，与其联系的有若干个下属模块，各下属模块还可进一步引出更下一层的下属模块。从图 3.2 所示的树状结构中可以看出模块的层次关系是十分清楚的。模块 A 是顶层模块，如果算做 0 层，则其下属模块 B 和 C 为 1 层，模块 D、E 和 F 为 2 层，等等。

从图 3.2 中我们还可以发现树状结构的特点是：整个结构只有一个顶层模块；并且，对任何一个下属模块来说，它只有一个上级模块与之直接关联。

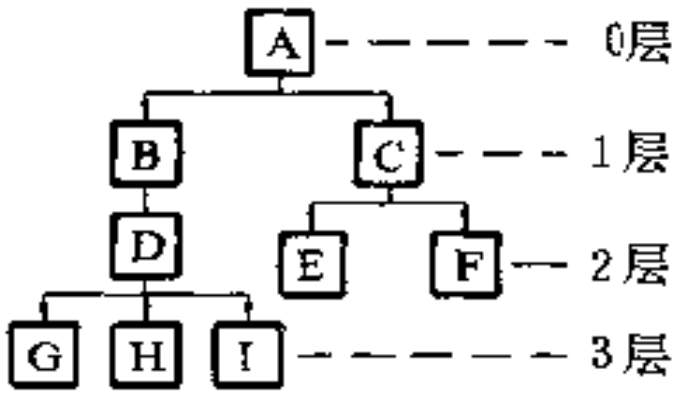


图 3.2　树状结构

网状结构的情况则完全不同，在网状结构中，任意两个模块之间都可有双向的关系。由于不存在上级模块和下属模块的关系，也就分不出层次来。任何两个模块都是平等的，没有从属关系。图 3.3 给出了网状结构的两个例子。在(a)图中，形式上模块 A 处在较高的位置上，它引出 B、C 和 D 作其下属模块。但我们仔细分析就会发现，B 也是 C 的下属模块，并且 C 又是 D 的下属模块。这都违背了上述一个下属模块只能有一个上级模块的层次规则，也就无法构成层次关系。在(b)图中给出了典型的网状结构。该图中五个模块之间都建立了平等的双向联系。

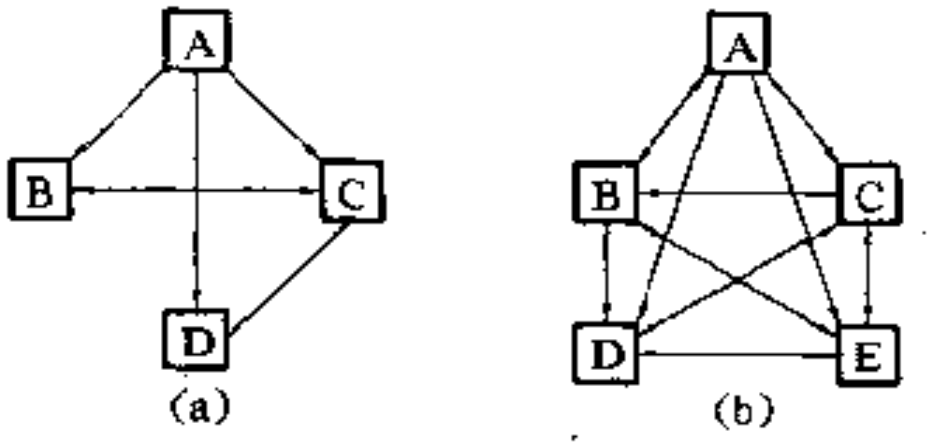


图 3.3　网状结构例

分析两种结构的特点之后，我们可以看出，对于不加限制的网状结构，由于模块间相互关系的任意性，使得整个结构十分复杂，处理起来势必引起许多麻烦。这样也就和原来划分模块，为便于处理的意图相矛盾。显然，网状结构中模块间的复杂关系抵消了模块划分带来的好处。因而在软件开发的实践中，人们宁愿采用树状结构，而不采用网状结构。

(2) 结构图：

结构图是精确表达程序结构的图形表示方法，它能清楚地反

映出程序中模块间的层次关系和联系。与数据流图反映数据流的情况完全不同，结构图反映的是程序中控制流的情况。结构图以特定的符号表示模块、模块间的调用关系和模块间的通讯。结构图中主要内容有：

① 模块：以矩形表示其中标有模块的名字，也可以在矩形内简要地指明模块的功能或是功能名的简称。对于已定义的（或已开发的）模块，则以双纵边矩形表示（见图 3.4）

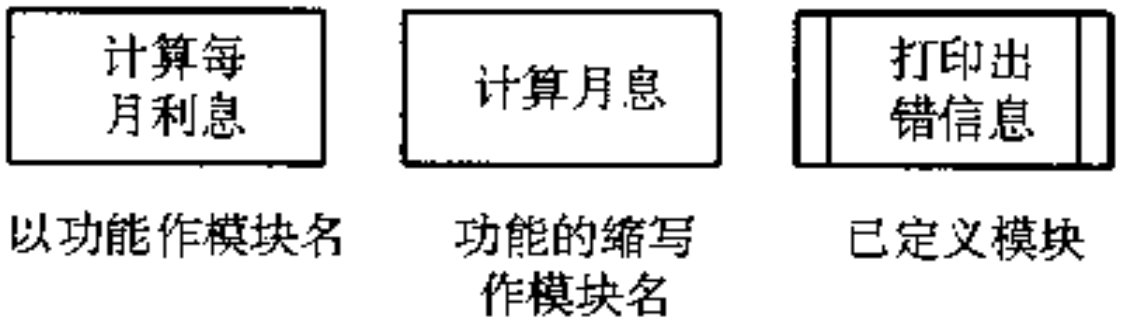


图 3.4　模块的表示

② 模块间的调用关系：两个模块，一上一下，以箭头相联，我们把上面的模块看作调用模块，箭头指向的模块看作是被调模块（图 3.5）。多重的模块调用自然构成了多层的结构图。

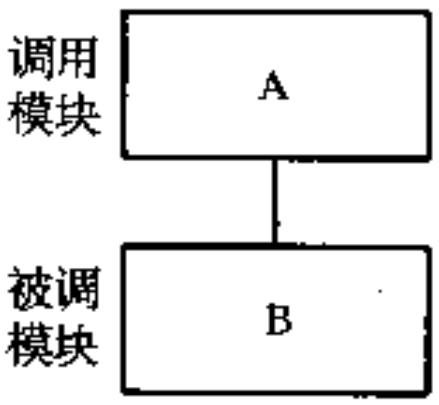


图3.5　模块间的调用关系表示

③ 模块间的通讯：在调用模块和被调模块之间总会有信息相互传递。通常所传递的信息可分为两种：

·二值控制信号：只表明是或否（肯定或否定）的两种状态。在调用箭头旁用标有信号名的符号●→ 来表示。

·数据：一般的信息。在调用箭头旁以标有数据名的符号●→ 来表示。以图 3.6 为例，学校教务管理中，在查询学生成绩的模块 A 工作时，给出某一学生的学号。调用查找学生姓名的模块 B，B 工作的结果首先送出学号是否有效的信号，然后给出学生姓名。有的结构图对这两种信

息不加区别，在图中均以注有信息名的箭头→来表示。

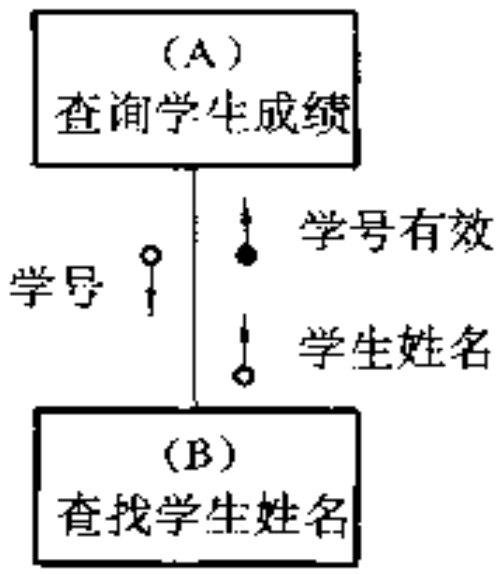


图3.6 模块间通讯的表示

④两个辅助控制符号：当模块 A 有条件地调用模块 B 时，在箭头的起点标以菱形。模块 A 反复地调用模块 C 和 D 时，另加一环状箭头。请参看图 3.7。在结构图中条件调用所依赖的条件和循环调用的循环控制条件通常都无需注明。

为了理解结构图的全貌，以下结合打印报表的数据处理问题举一实例。图 3.8 含有四层模块。整个报表加工的步骤是，首先顶层的主模块得到控制，然后沿着结构图的左枝下到底层模块(读入信息)，再沿左枝向上逐个控制执行右结点，直至回到顶模块。在执行完中间模块(计算)以后，沿右枝向下，经左分枝到底层(打印)，再执行中分支和右分支(都是打印)，最后重新回到顶模块。这个控制执行过程如同图 3.9 所示的树结构遍历顺序。请注意，不要混淆图 3.9 的控制执行顺序与图 3.8 中模块间传递的信息和控制信号的区别。

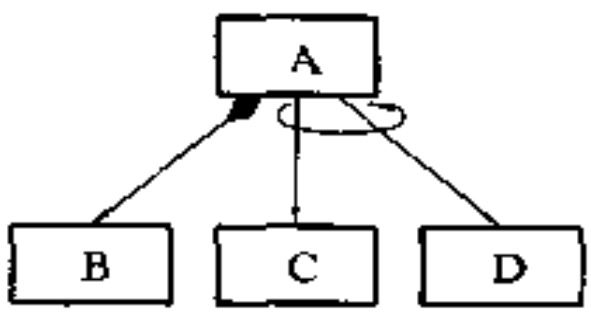
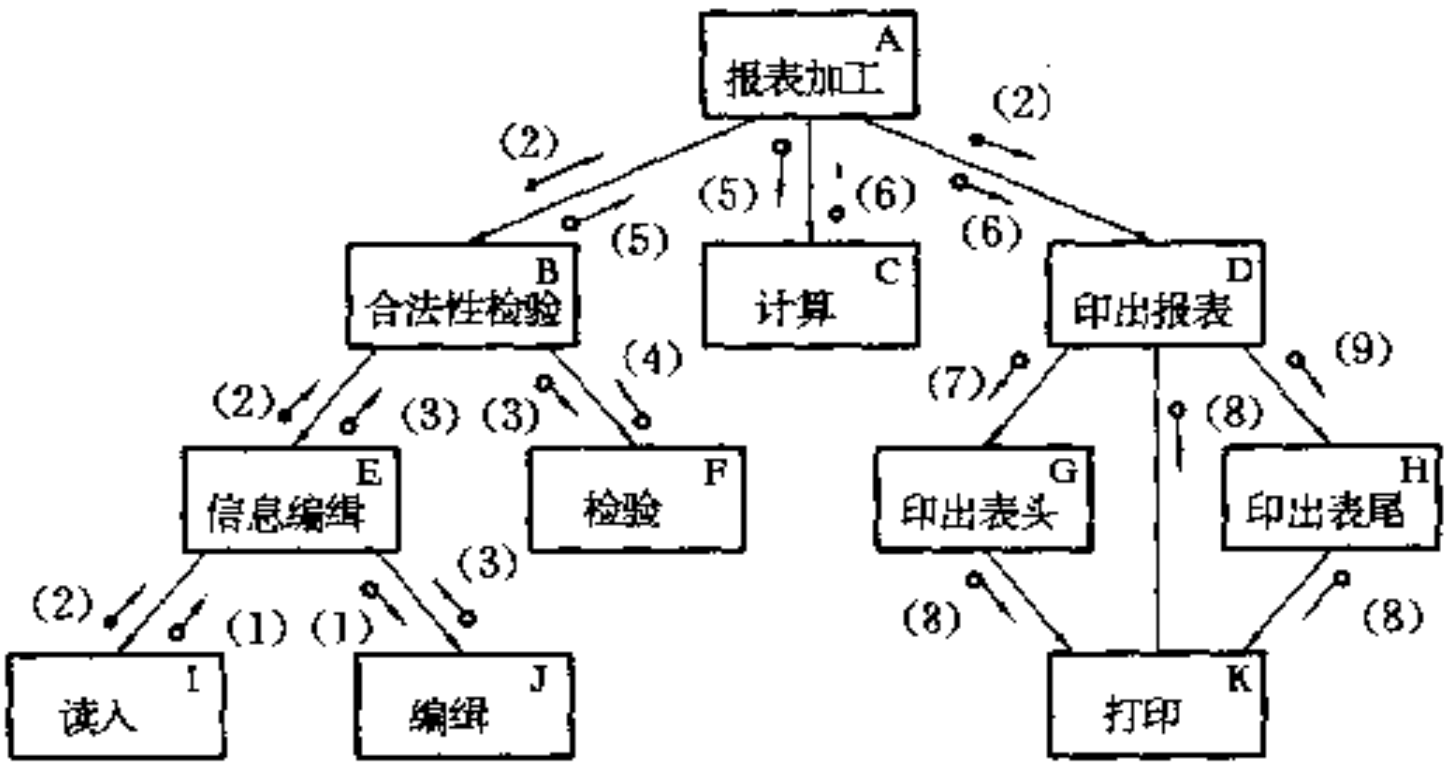


图 3.7 条件调用和循环调用在结构图中的表示

一般说来，结构图中可能出现以下四种类型的模块：

·传入模块——从下属模块取得数据，经过某些处理，再将其传送给上级模块(图 3.10 中的(a))。



信息流	
号码	信息流名
1	读入信息
3	已编辑信息
4	已检验信息
5	合法信息
6	结果信息
7	日期
8	信息行
9	最后结果

控制流	
号码	控制流名
2	最后输入信息

图 3.8 报表加工结构图

·传出模块——从上级模块取得数据，进行某些处理，传送给下属模块(图 3.10 中的(b))。

·变换模块——从上级模块取来数据，进行特定处理后，送回原上级模块(参看图 3.10 中的(c))。

·协调模块——对其下属模块进行控制和管理的模块。

此外，值得我们注意的是，结构图着重反映的是模块间的隶属关系，即模块间的调用关系和层次关系。它和程序流程图(常常称

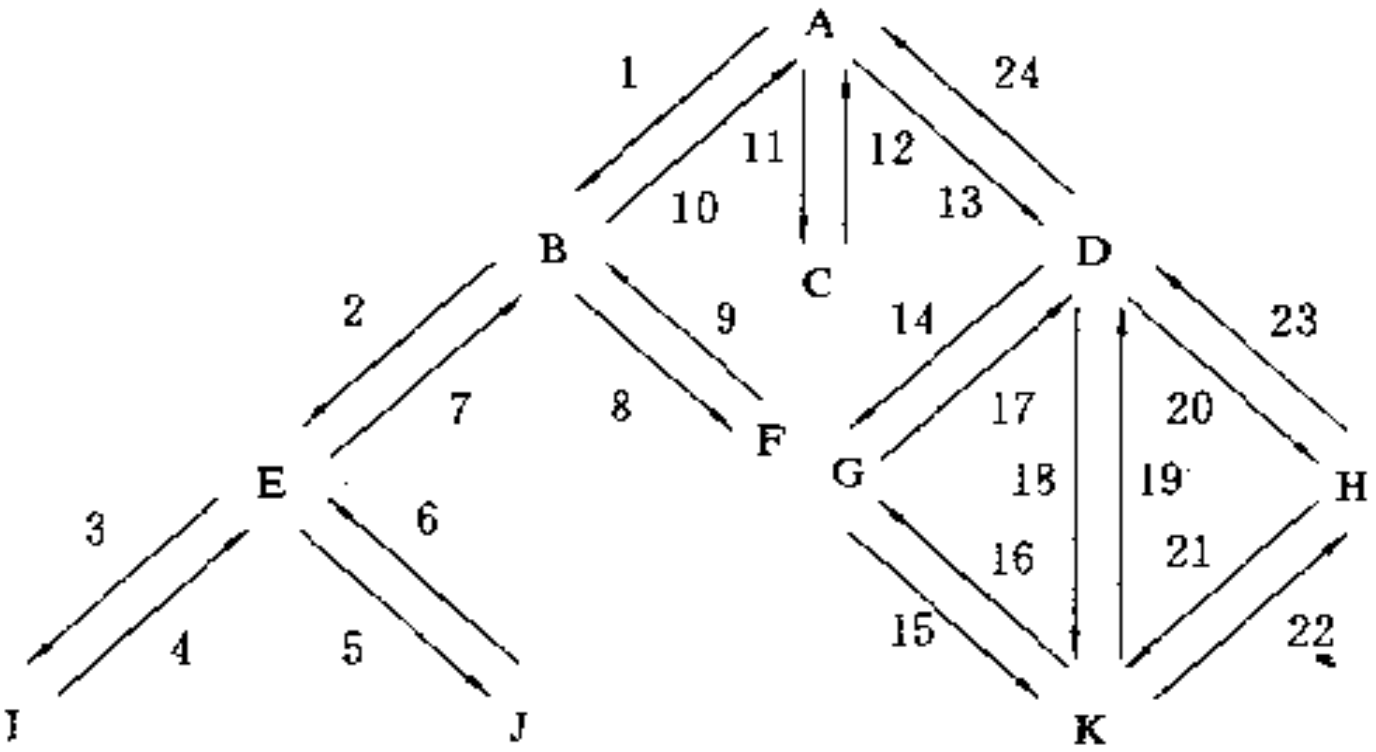


图 3.9　报表加工程序的控制遍历顺序

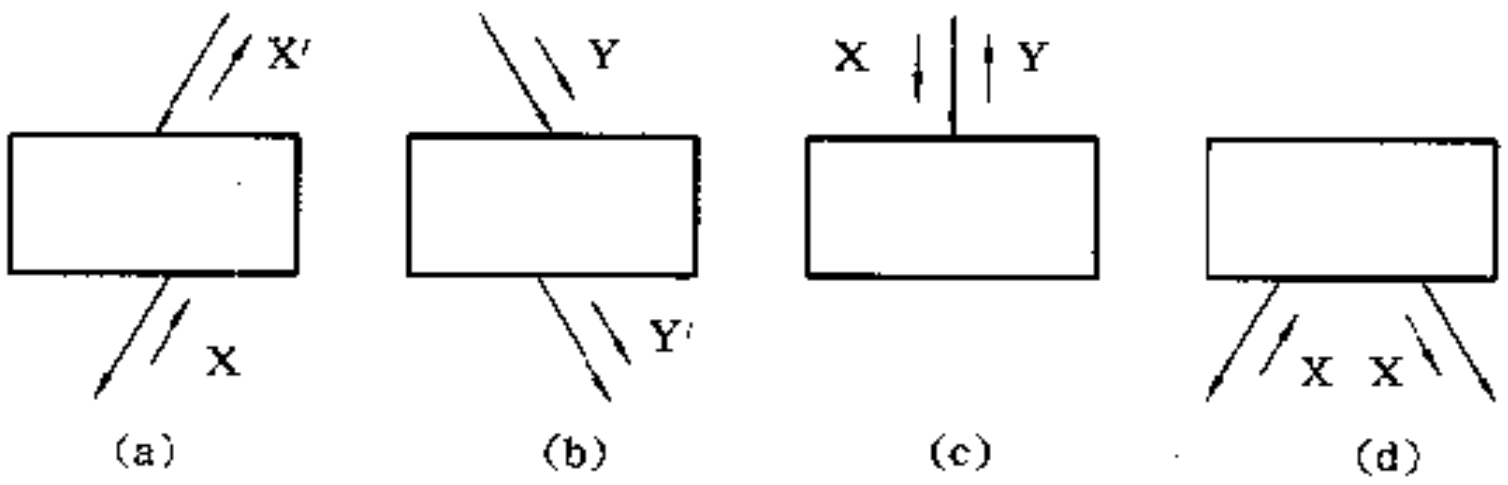


图 3.10　结构图中的四种模块类型

为程序框图)有着本质的差别(图 3.11)。程序流程图着重表达的是程序执行的顺序以及执行顺序所依赖的条件。粗略地说,结构图着眼于软件系统的总体结构,它并不涉及模块内部的细节,只考虑模块的作用,以及它和上、下级模块的关系。而程序流程图则用来表达执行程序的具体算法,我们将在下一章讨论详细设计的表达方法时仔细研究它。一些习惯于使用流程图编写程序的人,往往在模块未作划分、程序结构的层次尚未确定以前,便急于用流程图表达他们对程序的构想。殊不知这就如同建筑一座大楼,在尚未决定建筑面积和楼层有多少时,已经开始砌砖了。这显然是不合适的。

这种不恰当的做法反映了程序人员尚未建立起软件开发的工程化的概念。事实上，跳过了概要设计这一步，不考虑好程序结构的总体设计，即使把程序写了出来，也将给后面的开发工作以至维护工作带来严重的不良影响。

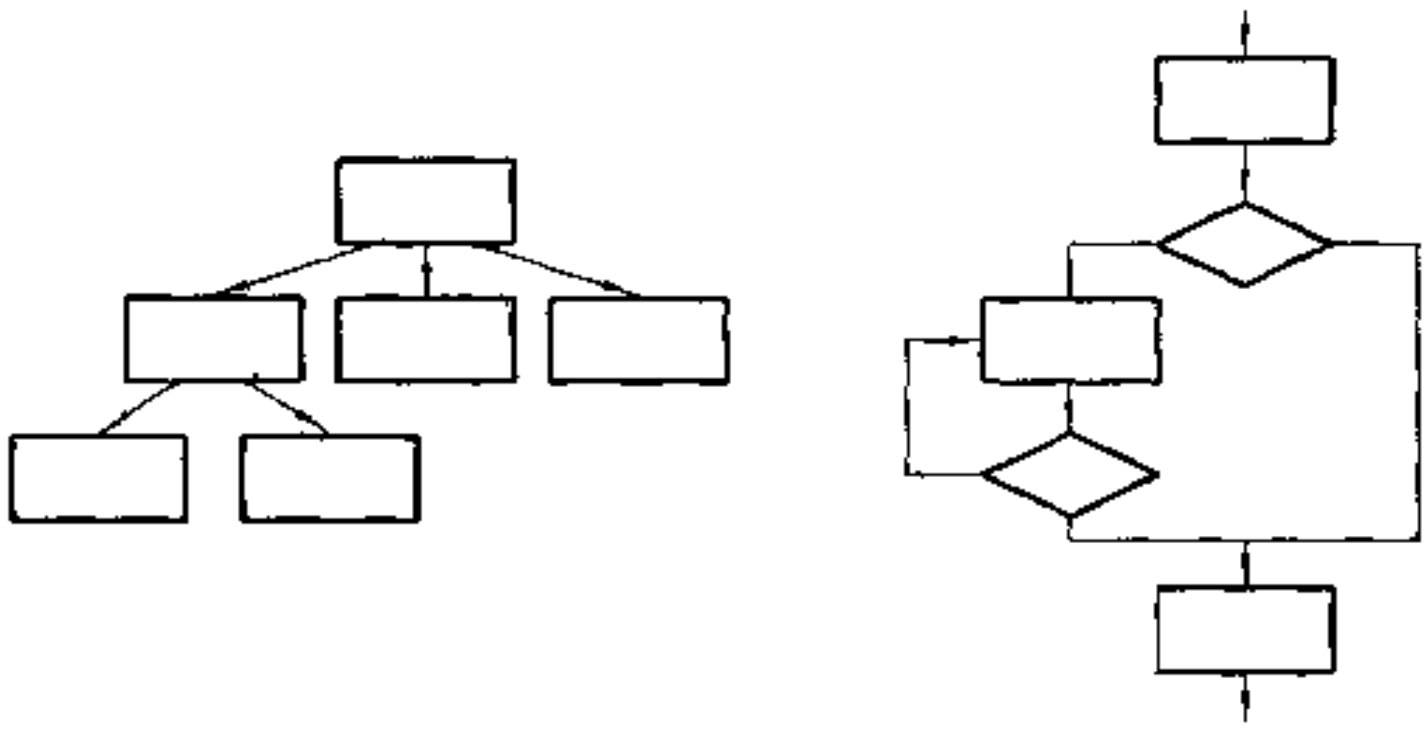


图 3.11　结构图与程序流程图比较

(3) 结构图的形态特性：

在树状结构图中，某些形态特性需引起我们的注意：

① 模块间以箭头表示的联线，在一般情况下也可以用直线代替。在图 3.12 中，图(a)和图(b)代表着相同的结构，也就是允许把箭头简单地画为直线。

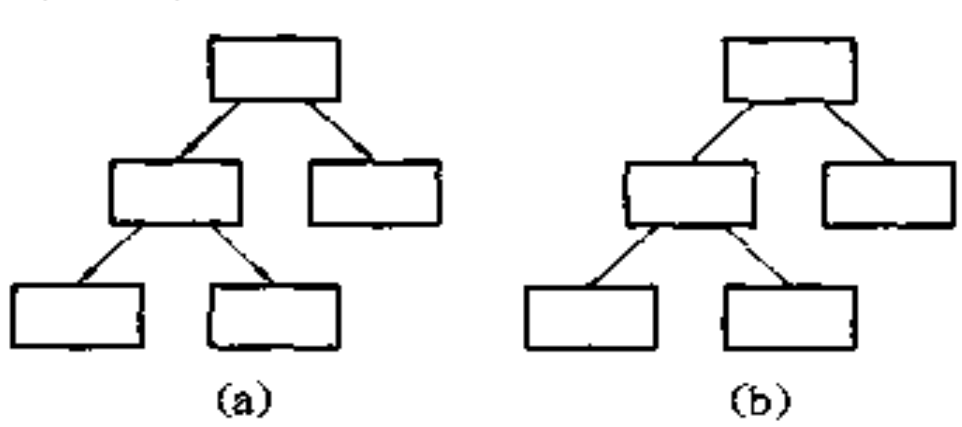
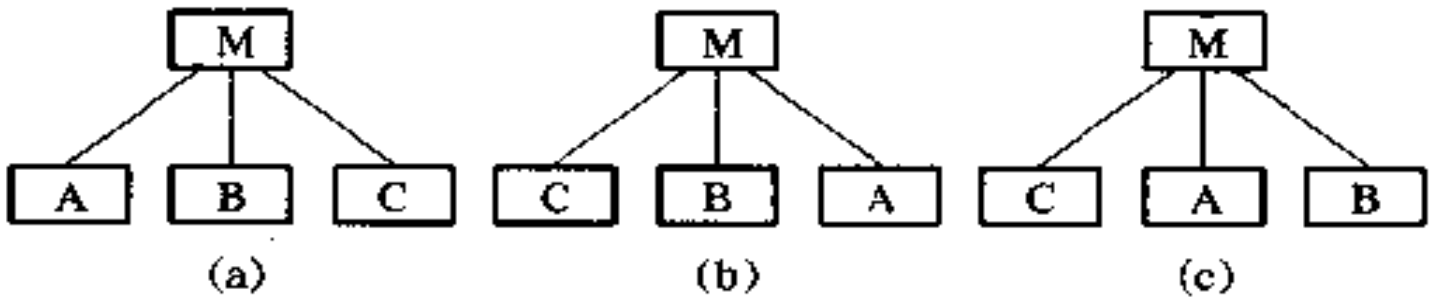


图 3.12　结构图中模块间联线与箭头同义

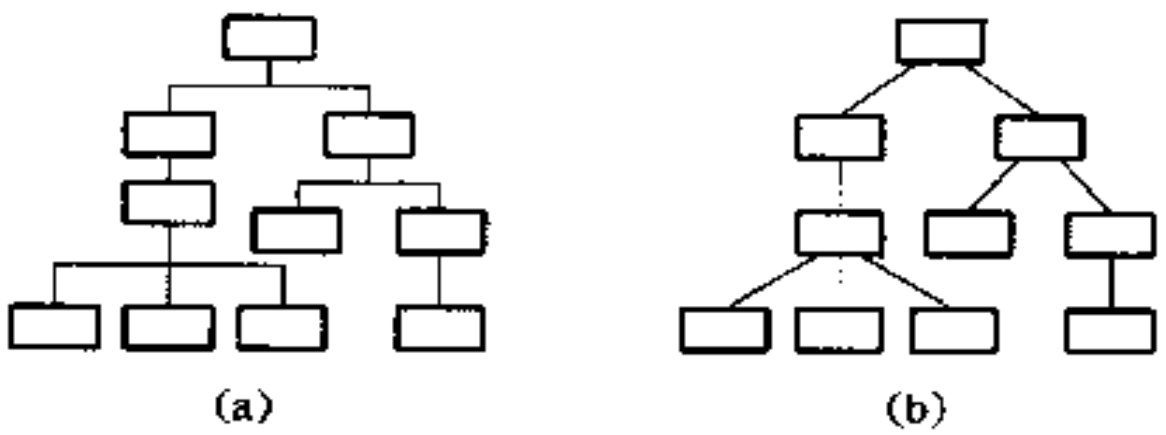
② 一个模块的多个下属模块在结构图中所处的左右位置是无关紧要的。比如，在图 3.13 中，图(a)、(b)和(c)是等价的，它们代表着同一个程序结构。但如果对下属模块的调用次序不是任意的，例如，必须按 A、B、C、的次序调用下属模块，最好是采用(a)图的形式，因为人们习惯于从左向右读图。



(a)　　(b)　　(c)

图 3.13　同一结构图的几种画法

③ 上级模块和下属模块之间的调用线，使用斜线和水平、垂直线具有相同的含意，因此 3.14 中(a)和(b)是等价的。



(a)　　(b)

图 3.14　结构图中模块间联线同义

④ 结构图的深度：在多层次的结构图中，其模块的层次数称为结构图的深度。如图 3.14 中的结构图，其深度为 4。结构图的深度在一定意义上也能反映出程序结构的规模和复杂程序。对于中等规模的程序，其结构图深度约为 10 左右。而大型程序，具有几十层的结构也是不足为奇的。

⑤ 结构图的跨度：结构图中同一层模块的最大模块数称为结

构图的跨度。结构图 3.14 的跨度为 4。

⑥ 模块的扇出和扇入：一个模块直接控制下属模块的个数称为该模块的扇出(也称为模块的跨度)。图 3.15 中(a)图 M 模块的扇出数为 7，(b)图中 N 模块的扇出数为 2。大的扇出数意味着需要过多地控制和协调下属模块，增加了问题的复杂性。比较恰当的扇出数是 2 至 5。在出现过高的扇出数时，完全可以通过增加层次的办法给予解决。例如，我们把图 3.15(a)的模块 M 多扇出结构改造成(c)图的形式。这样，通过增加两个中级模块 M_1 和 M_2，降低了扇出数。

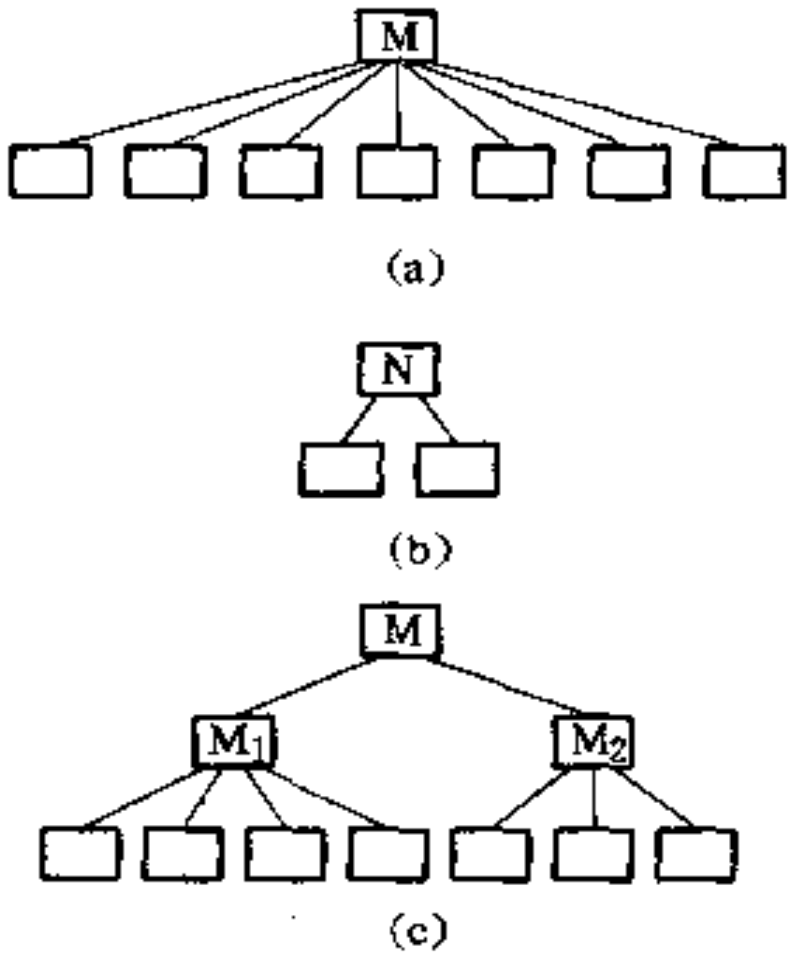


图 3.15　多扇出及其改造

在结构图中，多个上级模块可有一个下属模块，该模块的上级模块个数称为扇入数。图 3.16(a)的 P 模块，其扇入数为 6。与前述高扇出的情况类似，我们也可通过增加中间模块的办法减少扇入数。例如，图 3.16(b)便是增加了 P_1 和 P_2 这一层中间模块，而降低了模块 P 的扇入数。

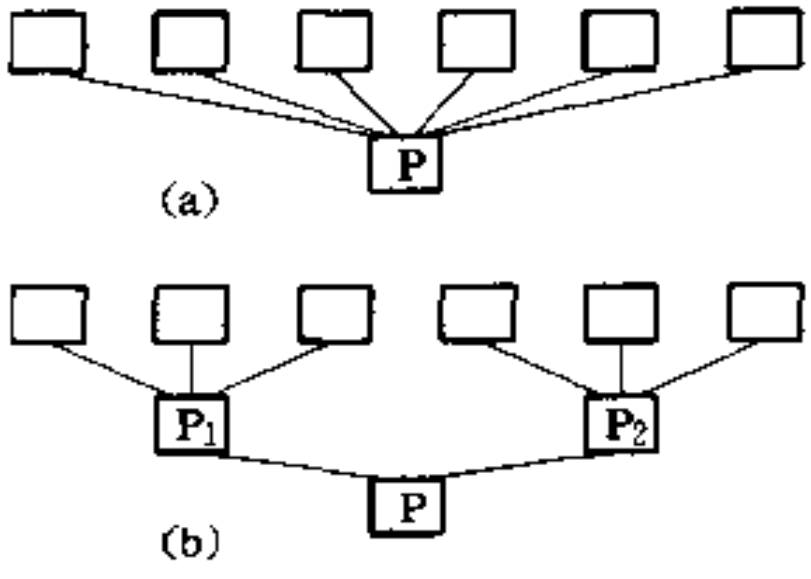


图 3.16　多扇入结构及其改造

尽管按树状结构来说，不允许出现多个上级模块控制一个下属模块的情况，但在程序的实践中，由于这种结构会给程序编写和测试等工作带来一些方便，所以在树状结构图中，也经常出现多扇入的结构。这时，其下属模块多作为公用模块使用。在图 3.8 的报表加工结构图中，右下端的打印模块便是这种情况，它的扇入数是 3。

(4) 结构图的程序含意：

前面我们已经把模块简单地说成是实现某些特定功能的程序段。其实，程序模块的概念自从人们有了程序设计知识就已经广泛地运用了。在程序设计中，我们非常熟悉子程序、分程序、函数等术语。从早期的汇编语言中的宏结构到新出现的高级语言，对这一概念都给予了充分的重视。诸如 SECTION(节)、SUBROUTINE(子程序)、FUNCTION(函数)、PROCEDURE(过程)以及 BLOCK(分程序)等都是为应用这一概念而设置的。实践表明，程序设计语言中提供了这些设施不仅大大方便了用户，而且为提高程序质量带来了许多好处。现在当我们考虑程序的总体结构时，完全有理由把它们也当作程序模块看待。

作为程序模块，它应该具有以下基本属性：

· 名字

· 明确的功能(关于这一点,我们还要在后面仔细讨论)

· 内部使用的数据或称局部数据

· 也其它模块相联系的输入或输出数据

· 实现其特定功能的算法

· 可被其上级模块调用,也可在其工作过程中调用下属模块(模块间的关系也将在后面作进一步讨论)。

这些基本属性大多可从图 3.8 的结构图实例中看到。

就整个结构图来说,它的程序含意是不难理解的。在结构图中,如果上下两个直接关联的模块仅限于调用和被调用的关系,同一层上的若干模块是并列关系,并且沿着结构图的纵向自上而下逐层调用,我们可按调用子程序的执行过程来理解各模块间的关系。例如,可以把图 3.8 给出的报表加工结构图表达为图 3.17 的

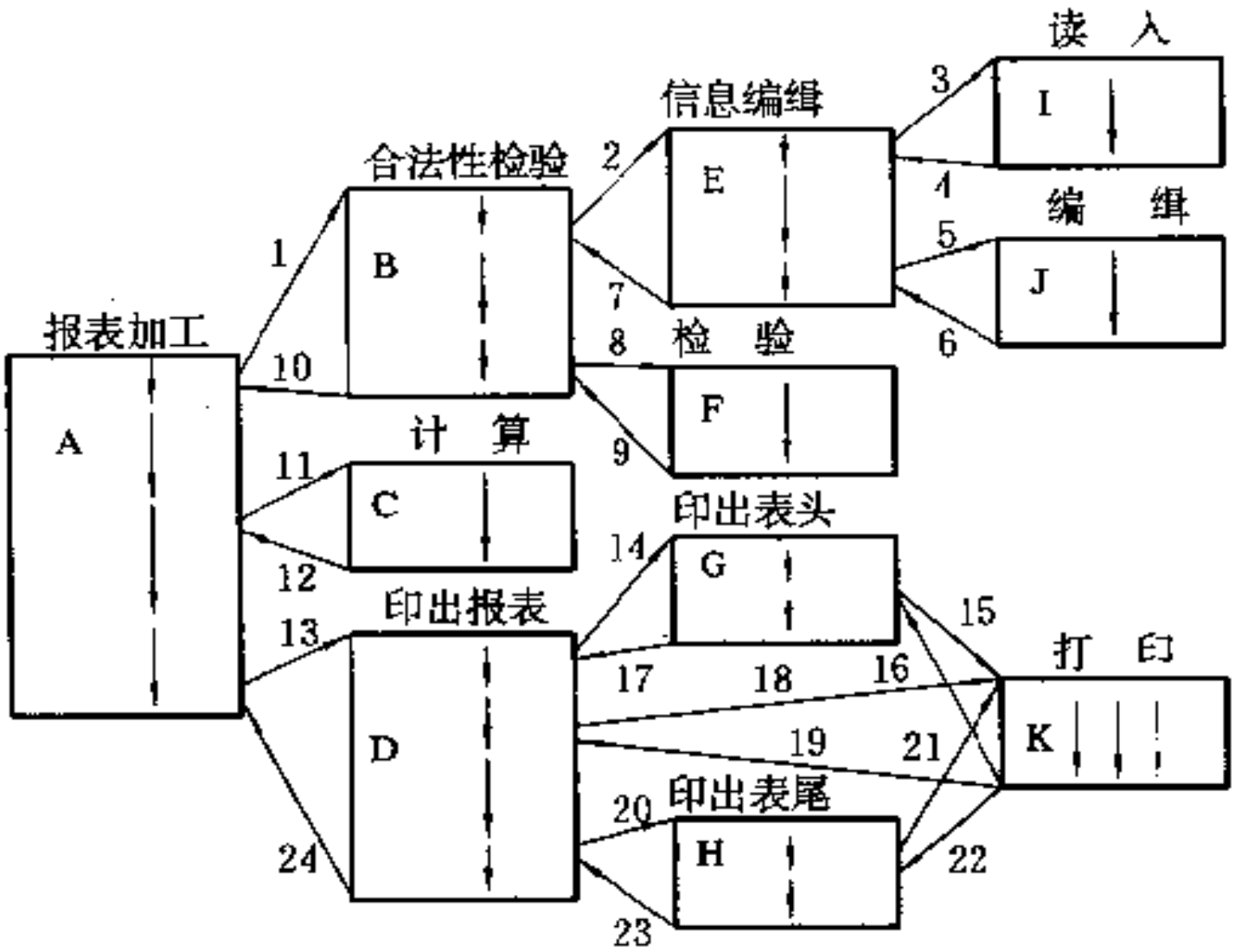


图 3.17 调用关系的另一表示

形式。这个图能更为直观地看出模块的多层次调用，同时也可看出各模块的执行顺序。请把这个图与表示报表加工程序的控制遍历顺序的图 3.9 作一对比。

现在如果让我们再来回顾高级语言提供的子程序、分程序、函数等设施，便可更清楚地看到，这些仅仅是为实现结构图中各个模块的算法服务的。从而也就更加明确了程序结构图在软件设计中的地位和作用。

3.3 程序内部的联系

程序内部的各个部分之间存在着多种多样的联系。当以结构图来表达程序结构时，我们最关心的是模块内部的联系和各模块之间的联系问题。我们把这两种联系分别称之为模块的内聚性(cohesion)和耦合性(coupling)。内聚性指的是在模块内部程序各部分之间的联系，耦合性则是指跨越在模块之间的联系。

在结构图上我们可以看到的是模块以及模块之间的联系。这时，实际上我们已经把模块当作具有某种特定变换功能的黑盒看待了。对结构图的讨论，大都属于模块间的联系问题。然而，应当注意到，这样做却是以明确每个模块功能为基础的。即是说以黑盒内部功能已经确定为前提的。以下将分别对模块的内聚性和耦合性作一般性的讨论。

(1) 模块的内聚性

划分程序模块可能遵循不同的原则，不同的划分方法会形成不同类型的模块内聚性：

① 功能内聚性

虽然对功能内聚性很难下一个严格的定义，但通常认为，具有功能内聚性的模块有着明确的职能。这种职能可以用简单的语句加以描述。模块中的各个部分都是为完成这一职能必不可少的组成部分。例如，某个模块的职能是“计算职工的月工资”。该模块的

各部分均以完成工资计算为目标协同工作，并且都是完成此目标不可缺少的部分，即彼此不可分割。为能做到这一点，模块的职能应该单一。也即在其职能的描述中不应有多个动词，不要用“和”、“首先”、“其次”等副词构成复杂语句。例如，某模块的职能被描述为：“计算职工的月工资和打印工资单”，便属于这一情况。较好的处理办法是，将其分解成“计算职工的月工资”及“打印工资单”两个模块。具有功能内聚性的模块，其职能明确、单一，例如：

计算三角形面积

打印错误信息

生成产值报表

检验单据的合法性

处理用户命令

等等。在图 3.8 所给的报表加工结构图中，各模块均有明确的、单一的职能，都是功能内聚性模块。

功能内聚性模块的优点是明显的。由于它的职能明确与单一，模块内部的联系非常紧密，与其外部的联系自然少而弱。所有出现在模块中的问题都可在本模块中得到解决，而不涉及到其它模块。它的明显优点包括：

- 便于理解　　• 便于修改
- 便于查错　　• 便于实现

② 顺序内聚性

我们在做模块划分时，也许会考虑把顺序执行的某些程序段放在一起，使其构成一个模块。该模块可能只含有不完整的部分职能，也可能含有多个职能。例如，有三个程序段：A、B 和 C，它们是按顺序依次执行的（见图 3.18）。如果 A 和 B 两段的职能是检验错误，C 的职能是打印错误。按顺序内聚性划分模块，可能有以下几种划分方法：

1）一个模块：包含 A、B 和 C 三个程序段

2）两个模块：A 和 B 组成一模块，C 为另一模块，或 A 为一模块，B 和 C 组成一模块

3）三个模块：A、B 和 C 分别当作模块

在以上的多种情况中，两个模块的前一种符合功能内聚性要求，这是应当推荐的，其它几种划分都因内聚性弱而应尽可能避免。

③ 通讯内聚性

如果模块各部分使用共同的数据，则构成通讯内聚性。例如，图 3.19 中模块 M 执行三个加工：生成日报表、生成周报表及生成月报表。这三个加工都需使用同一数据：日产量。

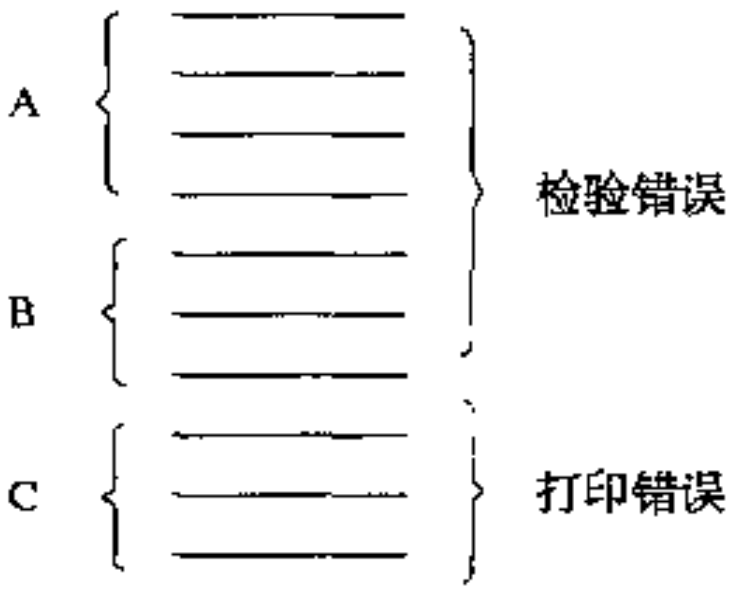


图 3.18　三个程序段完成两个功能

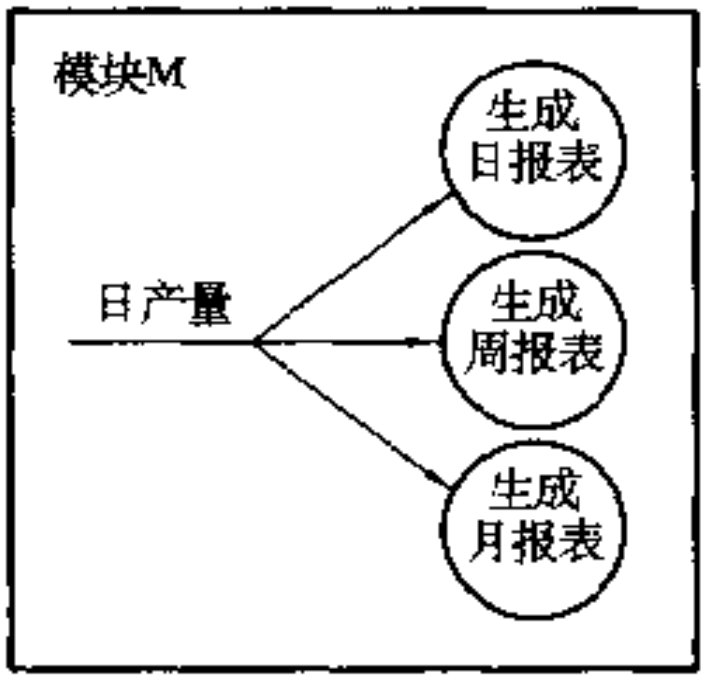


图 3.19　具有通讯内聚性的模块

④ 临时内聚性

有些模块内各部分之所以组合在一起是因为它们的执行时间决定的。例如，需要同时执行的一些操作。初始化一些变量、初始化寄存器或栈、打开或关闭一些文件以及检查某些量的极限值或标准值等都属于这类操作，实际上这些操作执行次序是不重要的，它们之间并没有逻辑上的必要联系，只是按类归并在一起而已。我们称这种模块为临时内聚性模块。

⑤ 逻辑内聚性

如果一个模块执行的是多个逻辑上相互关联的任务，它便是逻辑内聚性模块。例如，以“读入文件”命名的模块中可能包含多项工作，比如可能有读入主文件、读入用户文件等。表面上看来，这个模块有着很明确的职能，但由于这种模块处理对象不具体，或者说有多个对象，致使它的内聚性受到削弱。遇到这种情况，我们宁愿把它分解成职能明确，并且处理对象单一的多个模块，以增强每一个模块的内聚性。

判断哪些属于逻辑内聚性模块的简单办法是，从对模块的描述中只找到动词，却找不到明确、具体的对象。例如：

“进行事务处理”——并未指明哪些事务

“读入文件”——未说明什么文件

“执行全部输入和输出操作”——倒底哪些

“计算最大值或最小值”——究竟哪些值

在划分模块时，不主张构造逻辑内聚性模块的另一原因是，由于多个任务交织在一起，会给程序的修改带来许多实际困难。

⑥ 偶然内聚性

具有偶然内聚性的模块是由一些互不联系的程序段随机地拼凑而成。这样的模块其内聚性最弱，它表现为模块内各程序段之间可能毫无联系，完全可作任意的再分割。通常对这样的模块很难找到简单的描述来反映它所做的事。

有的程序人员如果片面地理解模块化的含义，他可能把含有多个功能的长程序（例如有五百行代码）机械地均分成十个模块（使每个模块包含五十行代码），从而构成偶然内聚性模块。这样做的结果会造成：

1）极大地削弱了模块的内聚性。一个原来完整的程序段，分割后有的功能可能跨越了多个模块。

2）多个功能也可能被包括在一个模块内。

3）由于所需的信息被划在其它模块内，频繁地相互引用成为

不可避免的现象。

4）写好的程序将难于看懂，也难于修改。这样做显然是很不恰当的。

还有一种情况，在作结构设计时，从多个模块中提出共同的部分，建立偶然内聚性模块。例如，模块 P、Q、R 和 S 本来没有什么联系，但每个模块中均有某一共同的程序段。于是会想到把共同的部分抽取出来，形成新的模块 X(参看图 3.20)。事实上，模块 X 并没有明确的职能。由于它的内容来自四个模块，必然和上述四模块有着依赖关系。这样做了以后，也许因为某种原因，其中一个模块，例如 P，要求它有所更动，这时 X 的更动势必影响到其它三个模块。由于这些缺点，在软件设计中，应尽可能避免采用偶然内聚性模块。

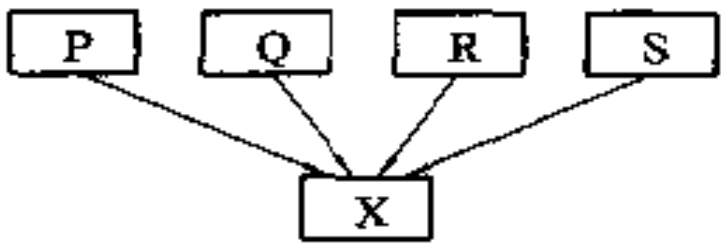


图 3.20　把四个模块的非功能程序段抽出构成模块

⑦ 各种内聚性的比较

前面的讨论已经表明，功能内聚性和偶然内聚性是两个极端。我们在划分模块时，应尽可能采用内聚性最强的功能模块，消除偶然内聚性模块。对于其它的内聚性也可以加以比较，按照内聚性的强弱，即按模块内部各部分之间联系的紧密程度进行排列。图 3.21 表明各种内聚性的排列次序。

(2) 模块的耦合性

模块的耦合性是程序模块之间联系紧密程序的度量。模块间联系越多，越紧密，其耦合性越强，同时也表明其独立性越差。在做模块划分时，可能遇到的模块耦合性有以下几种类型：

① 数据耦合

若某两个模块之间是调用关系，相互传递的信息以参数形式给出，并且传递的参数完全是数据元素，而不是控制元素，称这种关系为数据耦合。它是模块间耦合性最弱的一种形式。通常我们希望尽可能采用这种形式。

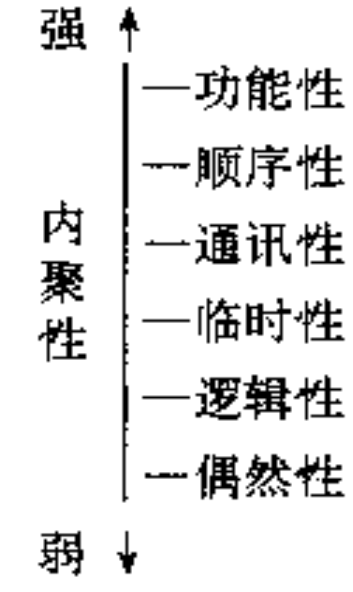


图 3.21　内聚性比较

② 公共耦合

一些模块工作时需引用共同的数据，这样的模块关系称公共耦合。在 FORTRAN 程序中大量使用的 COMMON 数据就为各程序块建立了公共耦合关系。由于这些模块被公用数据结构束缚在一起，对个别模块的修改和再利用必然带来许多不方便。

③ 控制耦合

如果一个模块明显地控制了另一模块的执行顺序，它们之间便是控制耦合的关系。例如，一个模块把控制执行顺序的开关值或控制变量值送入另一模块，以此影响另一模块的执行。对于这种耦合情况，发送控制信息的模块必须对接收控制信息的模块有十分具体的了解，一旦接收模块有了更动，势必会影响到这种控制关系。另一方面控制模块的修改也往往影响到接收模块。图 3.22 示出一模块向另一模块传送控制信号的实例。

④ 内容耦合

如果一个模块对另一模块的内容（包括程序段或数据）作直接的引用，也许通过非正常入口进入另一模块，或是对另一模块的内容（程序段或数据）作直接的修改，甚至两个模块共享一段代码，都构成了内容耦合关系。这种关系使得模块间的联系过分紧密，常常给后期的开发和维护工作带来不利的影响。

⑤ 各种耦合性的比较

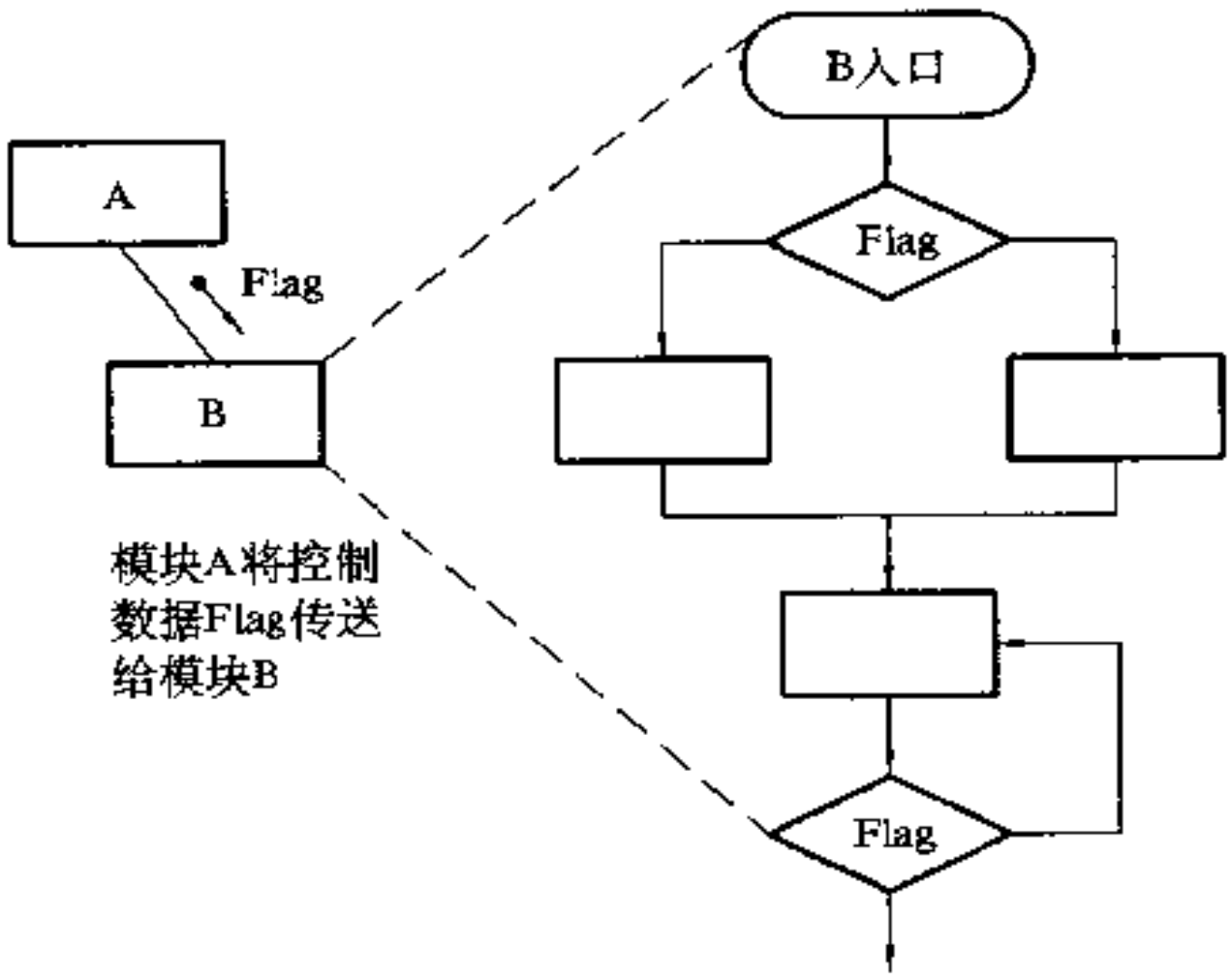


图 3.22　控制耦合例

耦合性既然表示了模块间的联系紧密程度，也就反映了模块间的独立程度。我们可把上述几种耦合性的强弱作一比较，虽然这里还给不出定量的度量标准，但却可以从比较中加深对它们的理解。图 2.23 示出各种耦合性的比较。

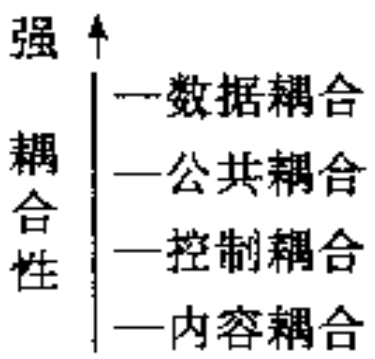


图 3.23　耦合性比较

(3) 模块化设计

软件设计的总目标是以较少的代价获得高质量的产品。实践表明，模块化设计能在相当程度上达到这一要求。这里将讨论按模块化方法进行设计的好处以及划分模块的原则。

① 模块化设计的优点：

1) 划分了模块，让每个模块完成单一的职能，把原来复杂的问题简化了，使复杂的多方面需求逐个得到满足。

2) 可以独立地进行模块的编码和测试，能够灵活方便地对这

些工作进行安排和组织，一个程序员可以完成若干个模块，也可以把各模块分配给多个程序员去完成，平行开展工作。

3）模块的划分把每个模块要解决的问题局限在有限范围之内，处理一个模块问题时不必考虑模块以外的问题，减少了出错的机会。即使出现了错误，在局部范围内也容易解决。

4）模块中一部分程序的修改，完全不影响模块以外的程序。极大地减少了产生修改副作用的可能。程序人员个人的工作差错，所影响的范围一般只限在模块以内，不会影响到全局。

5）可对关键模块采取特殊措施加以优化处理，以保证整个系统达到特定的要求。

6）使程序的复用成为可能，一个模块可多次使用，提高了软件产品的利用率。

7）已开发的程序易于理解，每个模块的职能明确，也就不难理解整个软件系统的结构和功能。

8）有利于估计工作量和开发成本。

② 模块的独立性

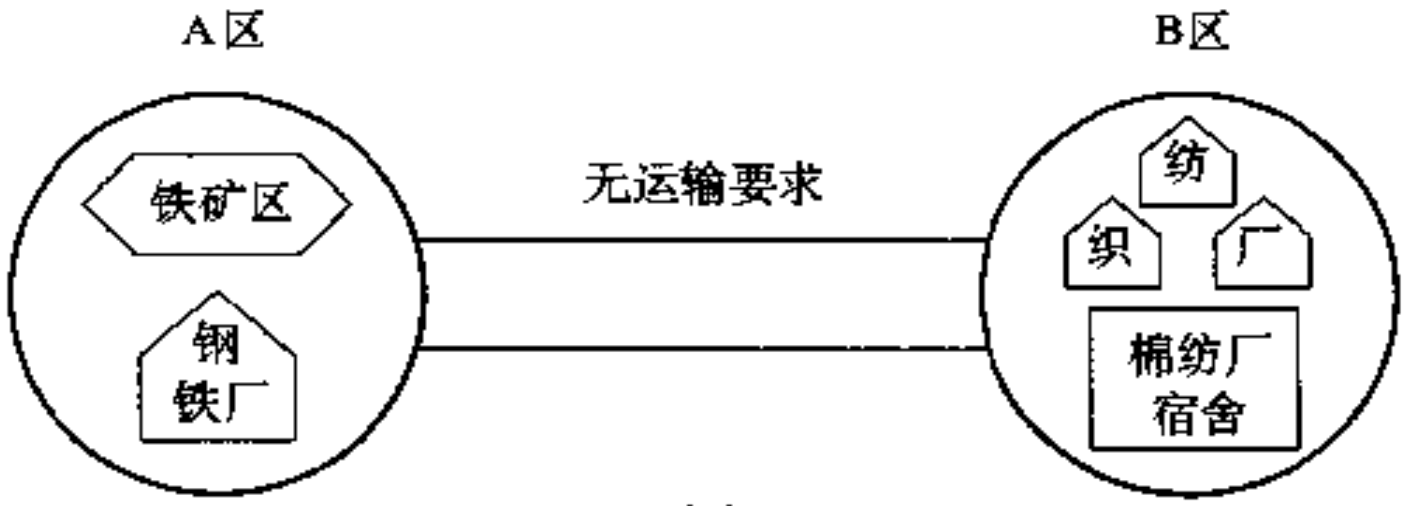
程序结构模块化的意义是无可怀疑的，但应该怎样划分模块，或者说划分模块应该遵循什么原则呢？这是在完成概要设计时的关键问题。指导模块划分的原则中应该特别重视的是模块的独立性。模块独立性的强弱可直接影响到软件的质量。

一般说来，究竟如何划分，并没有严格的、绝对的标准。好的模块划分方案也不是唯一的，同一个问题由两个设计人员考虑，可能得到不同的模块。只要这种划分是合理的，符合一定原则，就应该接受。因为问题本身是客观的，但如何划分则完全是主观的。然而这并不是说可以任意划分。这里完全可以举国家行政区划分作例子加以说明。我们考虑国家行政区的划分也是有原则的。比如，自然条件和地理位置、民族宗教以及历史因素等等。为什么在这里建立一个省，那里建立一个市，恐怕都有许多原

因，特别其中会包含着一两个具有突出意义的特征。也许有人会提出：为什么不把河北省和西藏自治区划为一个行政单位？回答可能是简单的。但如果真的这样划分了，若要提出反问：为什么要把河北省和西藏自治区划归一个行政单位？这个问题却是不容易回答的。因为恐怕找不出充分的理由能够说服人。

我们还可以举出另外的例子来说明问题。例如，某市有若干个区，其中 A、B 两区相距五十公里(参见图 3.24)，A 区有丰富的铁矿资源，并准备建一纺织厂，B 区计划建设钢铁厂和棉纺厂职工宿舍(a 图)，这样的规划方案显然会造成两区交通线上繁忙的运输：钢铁厂要把所需的铁矿原料不断的从 A 区运往 B 区，以供冶炼。棉纺厂工人则需早晚乘车往返。如果对这一规划作些调

(a)



(b)

图 3.24　两区的不同规划

整：把钢铁厂建在 A 区，使铁矿原料得到就地加工；棉纺厂建在 B 区的宿舍附近，工人便可就近上班，免去了往返之劳。这样调整的结果，在 A 区和 B 区的交通线上完全免除了这几个企业的大量运输要求(参看图 3.24(b))。显然，这样调整是合理的。现在让我们把 A、B 两区看作两个程序模块，调整以前模块内部的联系并不多：铁矿石和棉纺厂之间没有什么关系，钢铁厂和棉纺厂宿舍也是无关的。然而，模块之间却有着频繁的联系，一旦交通中断，几个企业的生产都直接受影响。调整以后，情况大为改观，A、B 两模块各自内部联系紧密，它们之间的联系相应地大大削弱了。如果使用前面的术语，调整以后加强了模块的内聚性，削弱了模块间的耦合性，最终达到了保持模块独立性的要求。

③ 划分模块的其它原则

除了上述加强模块独立性的要求以外，还有其它一些原则，可供划分模块时考虑：

1) 尽可能把与硬件相关的部分集中在一起，比如放在一个模块内或几个模块内。

2) 把有可能变动的部分尽量集中在一起，以利于确有变动时，方便处理。

3) 尽可能消除重复的工作，建立公用模块，以减少冗余，也减少了不必要的重复工作量。这对于后面的编码、测试以至维护工作都是十分有益的。

4) 恰当地掌握划分模块的大小。究竟划分多大的模块最合理，很难给出绝对的标准。但一般认为。程序最好能够写在一页纸内，或者说，程序行数在 50—100 的范围内是比较合理的。因为，同一个问题如果把模块划得很小，势必模块数量增多，显得十分琐碎。这就增加了模块接口的复杂性，也增加了花在调用和返回上的时间开销，降低了工作效率。另一方面，如果把模块划得过大，那将会造成测试和维护工作的困难。实际上，大的程序

块也是难于看懂的。

5）尽可能得到合理的程序结构图形态。掌握每个模块具有恰当的扇入数和扇出数是很必要的。正如本章 3.2 节所述，过多的扇入和过多的扇出都是要避免的。如果在开发项目中遇到这一情况，可采取重新调整模块的办法来解决。例如，图 3.25(a)表明模块 P 有六个上级模块，如果它不是公用模块，很可能包含多项功能，我们可以试着将它分解成 P_1、P_2 和 P_3 三个模块，其中 P_3 是 P_1 和 P_2 的公用模块。图 3.25(b)表明出高扇出模块 Q，如果不是实现多情形语句 CASE 的分类模块，也可将其分解成 Q_1、Q_2 和 Q_3。这样的分解既使模块功能单一化，又使模块的扇出减少。

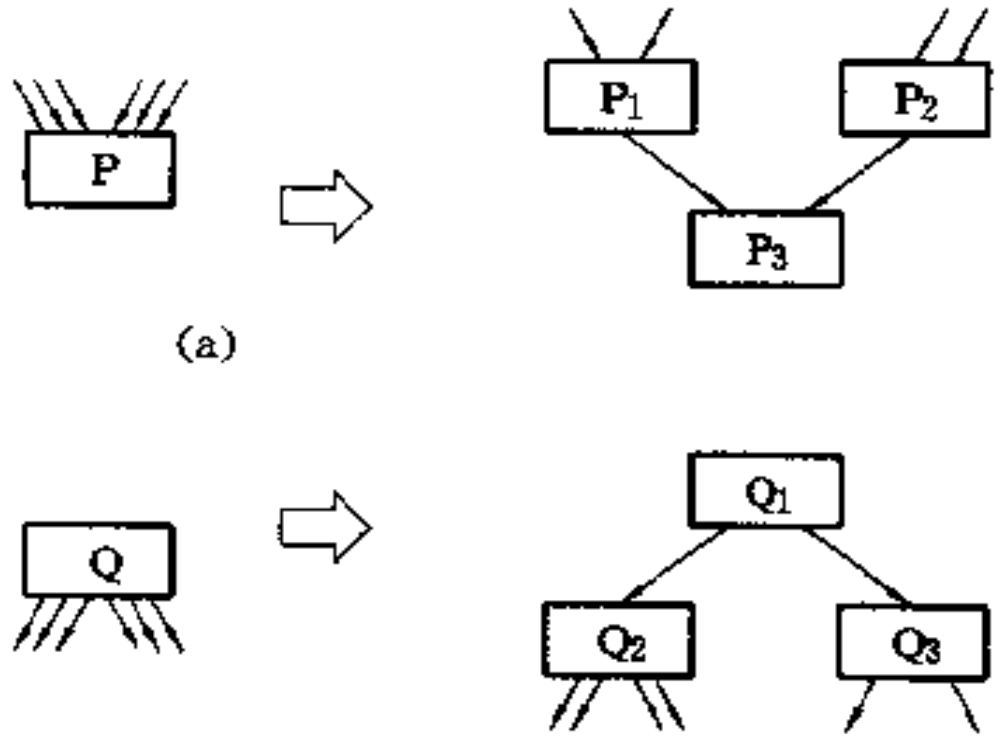


图 3.25　高扇入和高扇出的分解

3.4　结构化设计方法

结构化设计方法(Structured Design－SD)是由 IBM 公司 Constantine 等人提出的，是自顶向下进行软件系统的总体设计思想发展而成的。该方法使得数据流图可以向结构图进行系统地转换，因而可以和需求分析阶段所采用的结构化分析方法很好地衔

接。正是由于这一点，该方法也被称为面向数据流的设计方法。在概要设计阶段划分程序模块时，使用试探的方法解决块间联系和块内联系的问题，可以逐步求得较好的效果。此外，这一方法还能和编码阶段的“结构化程序设计”相适应，受到软件开发人员的欢迎。

① 使用结构化设计方法的步骤

采用 SD 方法进行概要设计的步骤是：

1）研究、分析以及审查数据流图。

首先应从软件规格说明书中弄清数据流加工的过程，这包括能够反映系统的外部界面的顶层数据流图，以及其下的各层数据流图。对于发现的问题应能及时得到解决。

2）根据数据流图决定问题的类型。

数据处理问题的两种典型的程序结构有两类：变换型和事务处理型(参阅本节下段)。针对两个不同的类型分别作不同的处理。

3）由数据流图推导出初始程序结构图。

以下我们还要针对两种典型的程序结构具体说明导出初始结构图的过程。

4）改进初始结构图。

使用试探法根据模块化原则改进初始结构图，直至得到满意的结构图为止。

5）修改和补充数据词典。

6）制定测试计划。

② 两类典型数据处理问题

在数据处理问题中有两类不同的典型，它们的数据流图和程序结构图都具有明显的特征。掌握了这些特征，便可以比较容易地建立起初始的程序结构图来。

1）变换型(Transform Type)。

变换型数据处理问题的工作过程大致分为三步，即取得数据、变换数据和给出数据(参看图 3.26)。这三步反映了变换型问题数据流图的基本思想，或者说是这类问题数据流图概括而抽象的模式。其中变换数据是数据处理过程的核心工作，取得数据只不过是为它作准备，给出数据则是对变换后的数据进行后处理工作。

图 3.26　变换型数据处理问题

这类问题的典型结构图如图 3.27 所示。看到这个图，读者可回忆起本章 3.2 节图 3.8 给出的报表加工实例。

在图 3.27 中，顶层模块首先得到控制，沿着结构图的左枝依次调用其下属模块，直至底层读入数据 A，然后，边作预加工边向上传送，以至形成逻辑输入 C，将其传送给主模块。在主模块

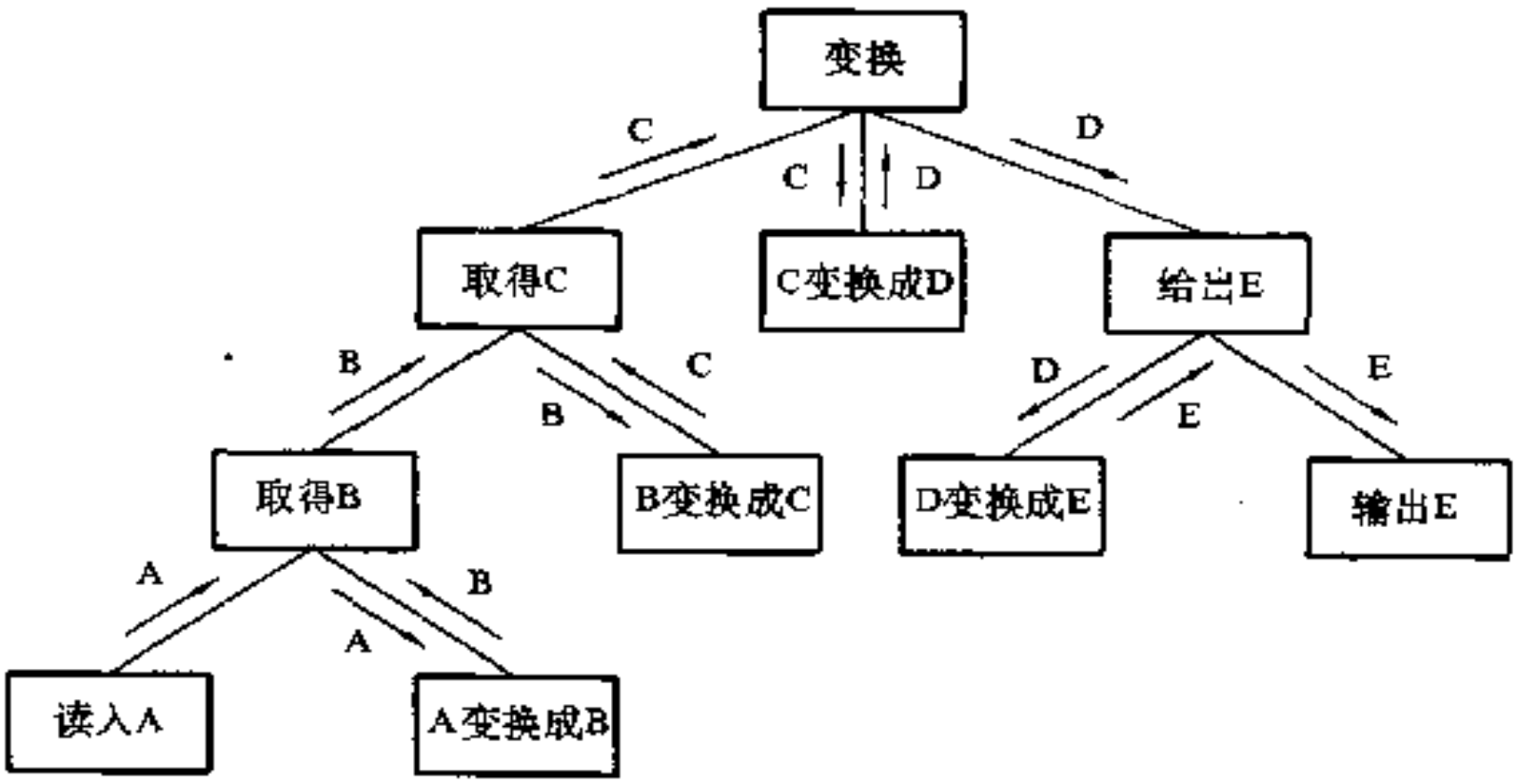


图 3.27　变换型数据处理问题的结构图

控制下进行中心加工，将 C 转换成 D，再边作后加工边下传，直至输出结果 E。显然，这种程序结构有着许多优点，每个模块都是功能模块，块内联系强，块间联系弱。模块之间传送的只是少量的数据型参数，界面清楚，易于理解。每个模块均可独立地进行编写、调试、修改和阅读。

让我们进一步分析图 3.27 给出的结构图。顶层模块是控制整个程序的变换(Transform)，记为 T。第二层左边模块(取得 C)是为变换作准备的源 (Source)结点，记为 S；二层的中间模块(C 变换成 D)是中心变换，也记成 T；二层的右模块(给出 E)是准备得到结果的汇(Sink)结点，记为 S。第三层的四个模块从左到右分别记为 S、T、T 和 S。第四层两个模块类似地记为 S 和 T。这样便可把结构图上所有模块的记号按结构图的样子，得出 S—T—S 符号树结构(参看图 3.28)。

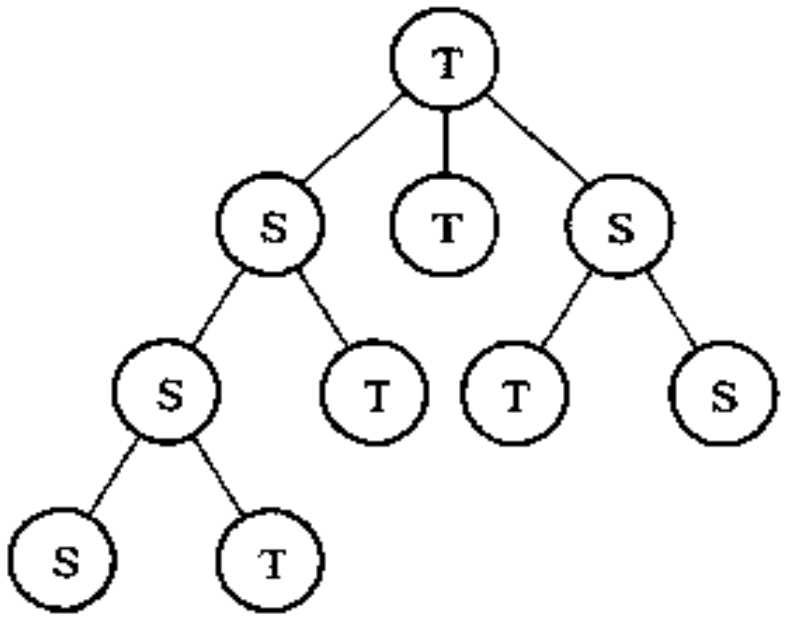


图 3.28　变换型问题的
S—T—S 符号树

2）事务型(Transaction Type)。

另一类典型的数据处理问题是事务型的。通常它是在接受一项事务的，根据事务处理的特点和性质，选择分派给一个适当的处理单元，然后给出结果。我们把完成选择分派的部分称为事务

中心，或分派部件(dispatcher)。这种事务型数据处理问题的数据流图可以图 3.29 为代表。其中，输入数据在事务中心 T 处作出先择，D_1—D_4 是并列的、供选择的事务处理加工。

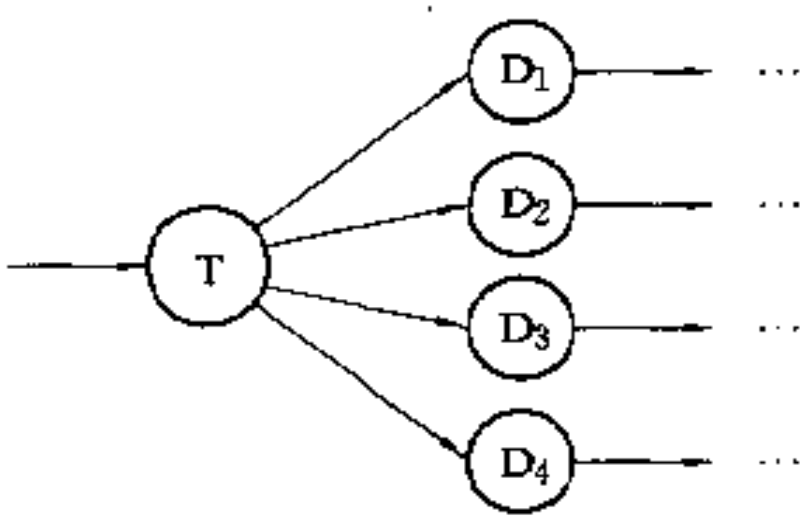


图 3.29　事务型数据处理问题

事务型数据处理问题的程序结构如图 3.30 所示。第二层的最左和最右模块分别负责数据的输入和输出，中间的 n 个模块是并列的，依赖于一定的选择条件，分别完成不同的事务处理。第二层以下的模块图中没有画出，这要根据具体问题作进一步的模块划分，但也可能出现公用的底层模块。

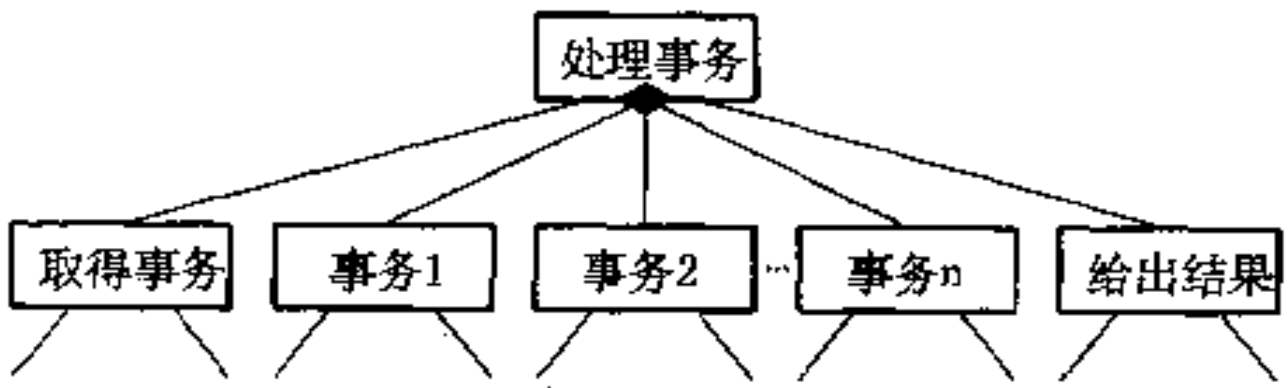


图 3.30　事务型问题的结构图

我们同样可以把事务型数据处理问题的结构图符号化。例如，把顶层模块记为 T；第二层的最左和最右模块分别是源结点和汇结点，都记为 S；中间一些事务处理模块也记为 T。我们就可构成图 3.31 所表示的符号化结构图。

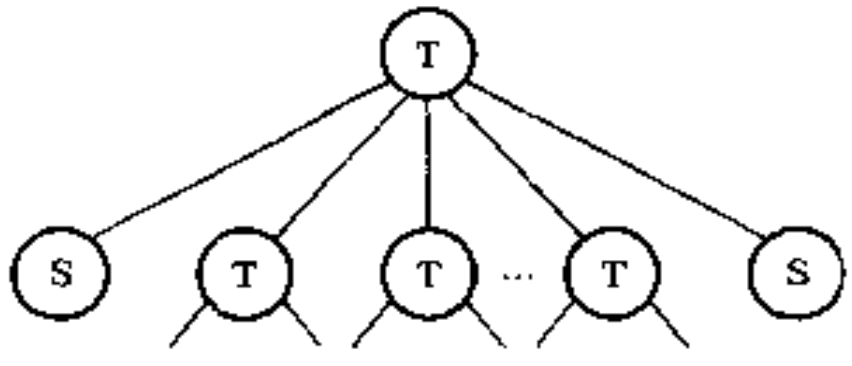


图 3.31　事务型问题的 S—T—S 符号树

③ 从数据流图导出初始结构图

在需求分析阶段我们曾经建立了数据流图，用以表达问题中数据流和加工之间的关系。在概要设计阶段则需要建立程序结构图，用以表示程序的总体结构。事实上。这两个图并不是毫无关系的。我们很自然地想到，能不能利用它们之间的关系，从数据流图导出结构图呢？哪怕是得到一个初始的结构图也好，因为对初始结构图逐步改进总能得到满意的结构图。以下针对两种典型的程序，分别讨论从数据流图导出结构图的问题。

1）变换型问题。

假定图 3.32 上半部是已给定的数据流图。分析这个数据流图我们看到，其中的“计算”是核心的数据处理部分，前面已称之为中心变换。它的左边“编辑”和“检验”均为给“计算”作准备的预变换。预变换以后，送入中心变换的数据流称为逻辑输入。中心变换送出的数据流称为逻辑输出 。从中心变换向右的部分均为给计算值作格式化处理的后变换。至此，对数据流图的分析主要是找到中心变换，这是从数据流图导出结构图的关键。有时中心变换能够很容易地被找到，例如，几支数据流的汇合处的加工常常就是中心变换。在它两端的数据流就是逻辑输入和逻辑输出。当中心变换不易找到时，可以从物理输入(也就是实际输入)数据流或从物理输出(即实际输出)数据流开始，逐步由数据流图的两个端点向中心推进。采取这种试探的方法最终总能够把中心变换找到。

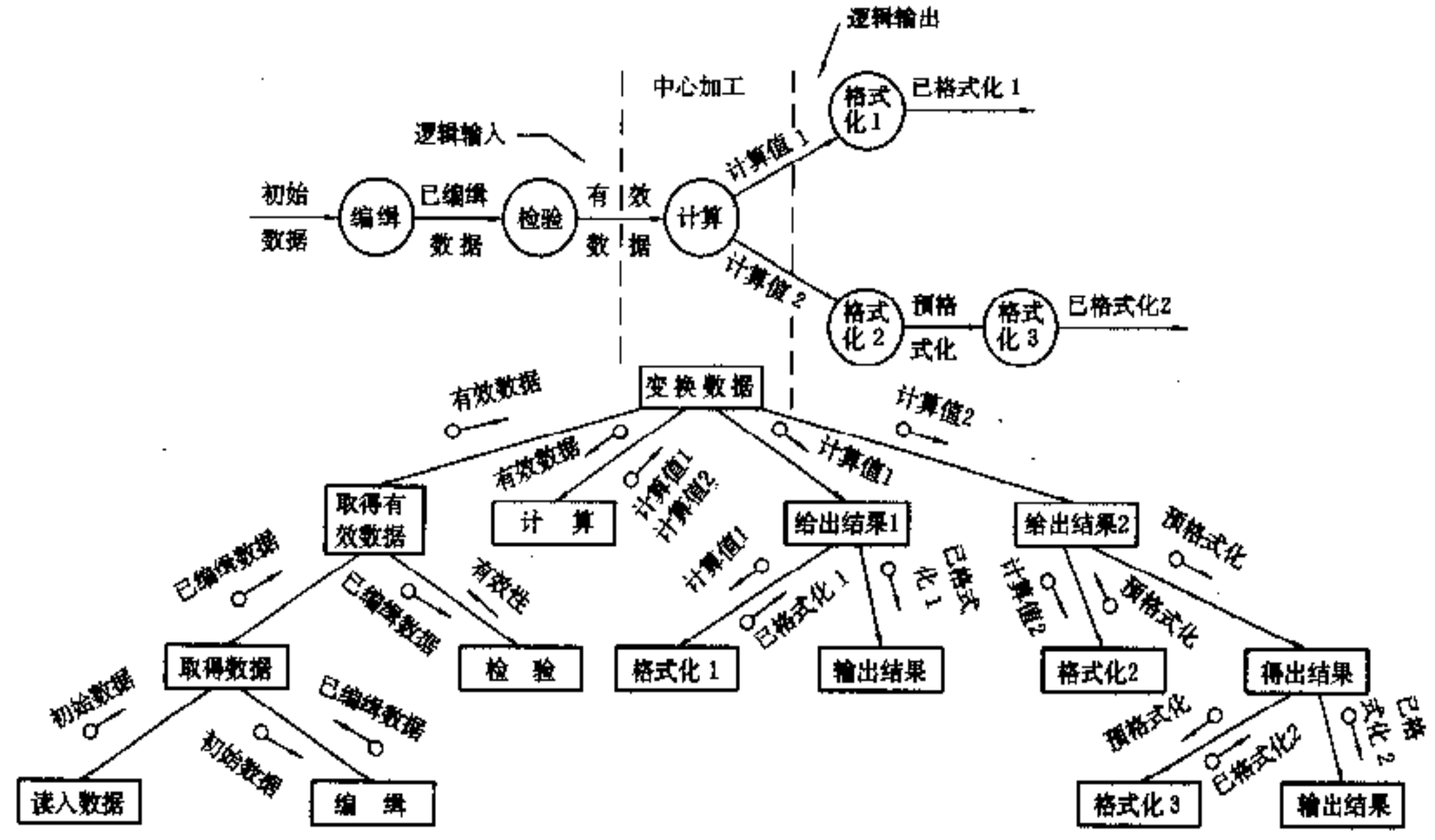


图 3.32　变换型问题的数据流图导出结构图

找到中心变换，便可确定结构图的顶层模块，接着继续分析数据流图的其它部分，逐步地自顶向下建立结构图的其它模块。

在图 3.32 中，把顶层的主模块对准中心变换，并给该模块命名程序的名称，因为主模块的功能是控制执行整个程序要做的工作。然后再考虑其下属模块的设计。

接着设计第二层模块。从变换型程序结构的符号图(参阅图 3.28)中我们知道，第二层可能有三种模块即：

- 向主模块提供逻辑输入的输入模块(S)
- 完成中心变换的模块(T)
- 从主模块送出两个逻辑输出的输出模块(S)

于是，可以得到第二层的四个模块，它们是："取得有效数据"、"计算"、"给出结果 1"及"给出结果 2"。请注意，结构图上模块间传递的数据与数据流图上的数据流有对应关系。至此我们已得到结构符号图(图 3.33)的上面两层符号。

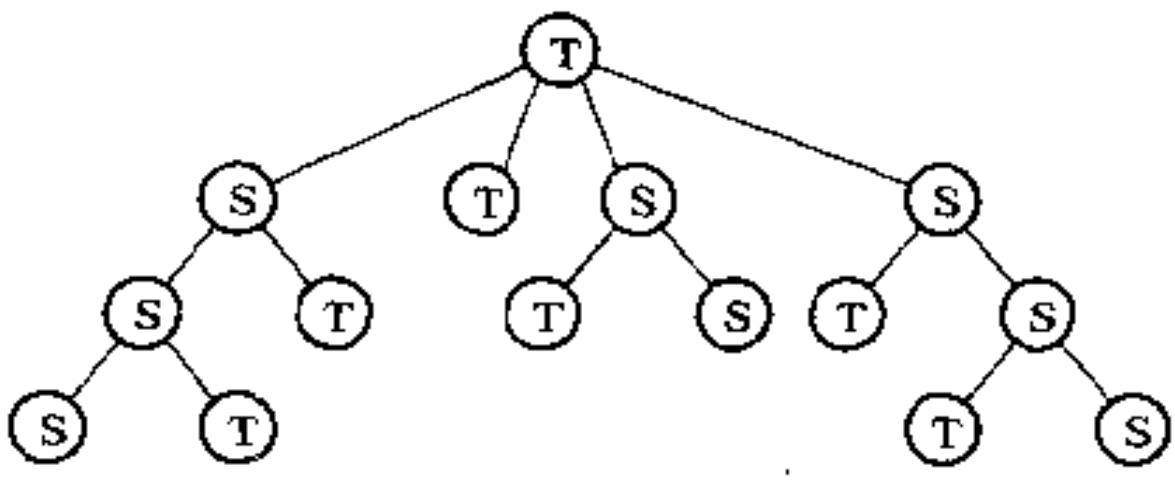


图 3.33　导出变换型问题结构图的雏型——S—T—S 符号树

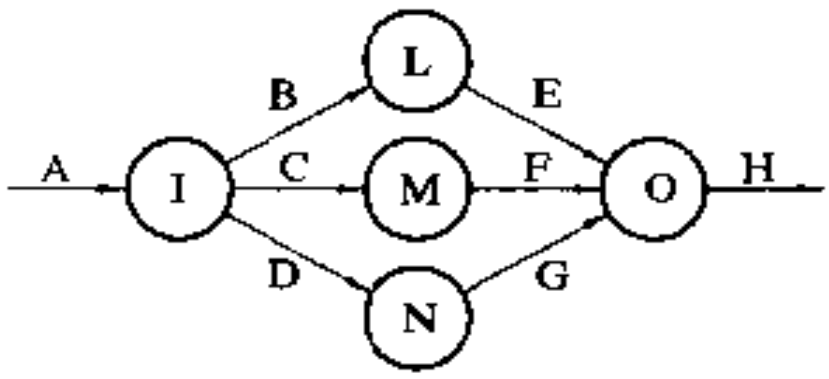
最后设计中下层模块。依照结构符号图的规律，输入模块(S)的下属模块应是 S—T 型，输出模块(S)的下属模块应是 T—S 型(图 3.28)。换句话说，输入模块的功能是向主模块提供数据。因而它应包含两种下属模块：取得数据模块及按要求完成变换的模块；输出模块的任务是按要求送出结果数据，它的下属模块应包含变换模块和输出结果模块。在中心变换无需下属模块的情形

下，我们得到六个第三层模块："取得数据"、"检验"、"格式化1"、"输出结果"、"格式化2"和"得出结果"。

以下结构图的设计完全类似，可按第二层设计的方法递归进行，直至出现物理输入和物理输出。中心变换的下属模块设计则应根据问题的具体情况考虑。我们在得到图3.32下面的结构图的同时，还得到了它的符号图(图3.33)。

2）事务型问题。

和变换型问题类似，导出结构图也需从分析数据流图开始，自顶向下地设计结构图。这里取图3.34上部的数据流图为例。在这个图中取得事务A后，按某一条件将其分派，完成L或M或N的处理，最后经O输出。



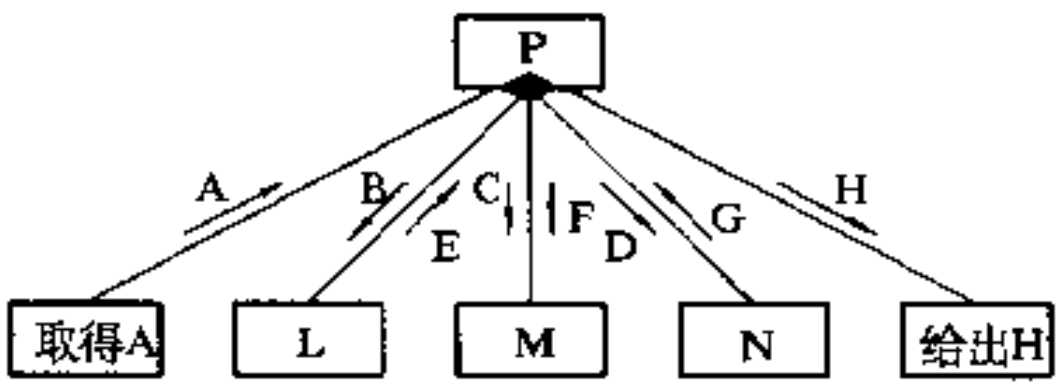


图3.34　事务型问题导出的结构图

首先建立主模块P代表整个加工。然后考虑第二层模块，这时我们应回想起图3.31所代表的典型事务型结构的符号图。第二层模块只能是三类：取得事务(S)、变换事务(T)及给出结果(S)。该数据流图三个加工：L、M和N是并列的，它们的工作应

由变换事务的模块来完成。因而在主模块的下沿以菱形引出三模块，分别完成 L、M 和 N 的工作，在它们的左、右两边则是对应于加工 I 和 O 的“取得 A”模块和“给出 H”模块。

如果加工 L、M、N 尚有更多的数据流加工要求，或者说还有它们的数据流子图，则可在结构图的第二层模块以下继续扩展，直至完成全部加工。

我们遇到的实际问题也许不完全属于变换型或是事务型的，但很可能是两者结合的。从数据流图上可看出兼有这两种类型的特点，例如，在事务型问题分派出的事务处理过程本身是变换型问题。对这样的问题，其处理方法参照前面的介绍，也是不难解决的。

第四章　详细设计的表达

如同建筑工程中的结构设计只考虑了建筑物的布局和框架一样，概要设计只解决了软件的总体结构设计问题。经过了概要设计，整个软件的构造，模块怎样划分，包括每个模块的功能及模块间的联系都已经确定下来。接着便要着手解决怎样实现每个模块功能的问题。这就是详细设计的任务。

也许有的软件人员以为，不经过详细设计，或者开发规模比较小的程序也可直接编写程序。但在多数情况下，特别是按软件工程的要求进行开发工作，详细设计这一步是不可缺少的。从软件开发人员方面看，在使用程序设计语言进行代码编写以前，需要对算法的逻辑关系进行详细分析，并给予清晰的表达，使它成为编码的依据。从用户和维护人员方面看，如果需要了解程序，弄清实现算法怎样达到模块的规定功能，直接读程序常常是很困难的，为了更容易地了解程序，也需要有一个详细设计的文件。

在详细设计阶段软件开发人员面临两个方面的问题，一个是决定实现每个模块的算法，另一个是如何精确地表达这些算法。自然，前一问题涉及到所开发项目的具体要求和对每个模块规定的功能。算法的问题不属于本书讨论的范围。后一问题需要给出适用的算法表达形式，或者说应提供详细设计的表达工具。本章将对目前最常用的几种算法表达工具作出简要的介绍。

程序流程图是我们早已熟悉的图形工具，但在使用中发现它有一些缺点，不能满足详细设计工作的需要。于是近年来又提出了一些新的算法表达工具，例如 N—S 图、PAD 图、PDL 语言、HIPO 图等。虽然这些工具在使用中各有自己的长处，但我们也会

发现它们的一些不足之处。遗憾的是，现在还没有一个十全十美的理想工具为人们普遍接受。

在设计这些算法表达工具时，大都注意到一个重要问题，就是应该考虑到服从结构化程序设计的原则。也就是希望采用这些工具表达算法时，可以方便地写出结构化程序。因此，要求每一种工具都能表达基本的控制结构。

4.1 程序流程图

程序流程图(Program flowchart)也称为程序框图，也许是软件人员最熟悉的一种算法表达工具。人们学习程序设计语言编写程序很早就使用它，至今仍然是我国软件人员最普遍采用的一种工具，并且仍然不失为初学者掌握程序设计方法的辅助手段。在需要了解别人开发软件的具体实现方法时，我们常常不得不去读他的流程图。实际上，这总比直接阅读程序方便得多。

流程图为人们普遍采用是因为它具有一些特有的优点。比如，它能把程序执行的控制流程顺序表达得十分清楚，看起来也比较直观，容易看懂。也许仅此一点便足以使得某些习惯使用流程图的人对它有着特殊的偏爱，不愿再去接受其它的新工具。

然而，需要指出的是，流程图确实也存在着一些严重的缺点，不应忽视。例如流程图所使用的符号不够规范，规定某些符号的使用方法不够严格、明确，常常使用一些习惯性的用法。特别是表示程序控制流程的箭头，使用的灵活性极大，若使用不当会使程序质量受到很大影响。这些现象显然是和软件工程化的要求背离的。为了消除这些缺点，我们应对流程图所使用的符号作出严格的定义，不允许人们随心所欲地画出各种不规范的流程图。

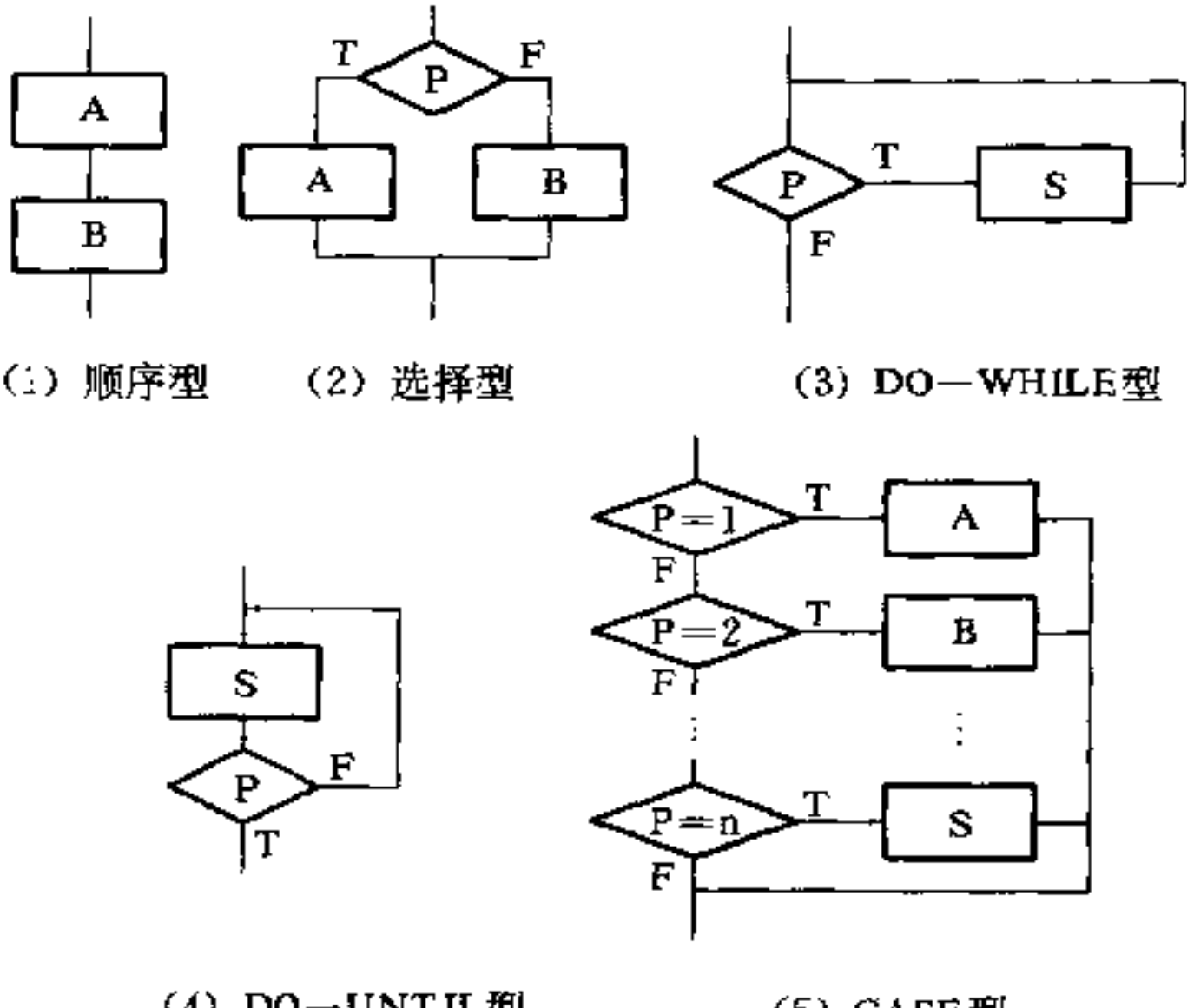
首先，为使流程图描述结构化程序，必须限制流程图只能使用图 4.1 中给出的控制结构，或者说，任一程序流程图应由图 4.1

所提供的五种基本控制结构组合或嵌套而成。这五种基本控制结构是：

（1）顺序型结构——含有多个连续的加工步骤；

（2）选择型结构——由某个逻辑条件式的取值决定选择两个加工中的一个；

（3）当(while)型循环结构——在控制条件成立时，重复执行特定的加工；



（1）顺序型　（2）选择型　（3）DO—WHILE型

（4）DO—UNTIL型　（5）CASE型

图 4.1　流程图的基本控制结构

（4）直到(Until)型循环结构——重复执行某些特定加工，直至控制条件成立；

（5）多种情形(Case)型结构——列举多种加工的情况，根据某控制变量的取值，选择执行其中之一。

限制基本控制结构的作法意味着不允许使用除此以外的控制

结构。作为上述五种控制结构相互组合和嵌套的实例，图 4.2 示

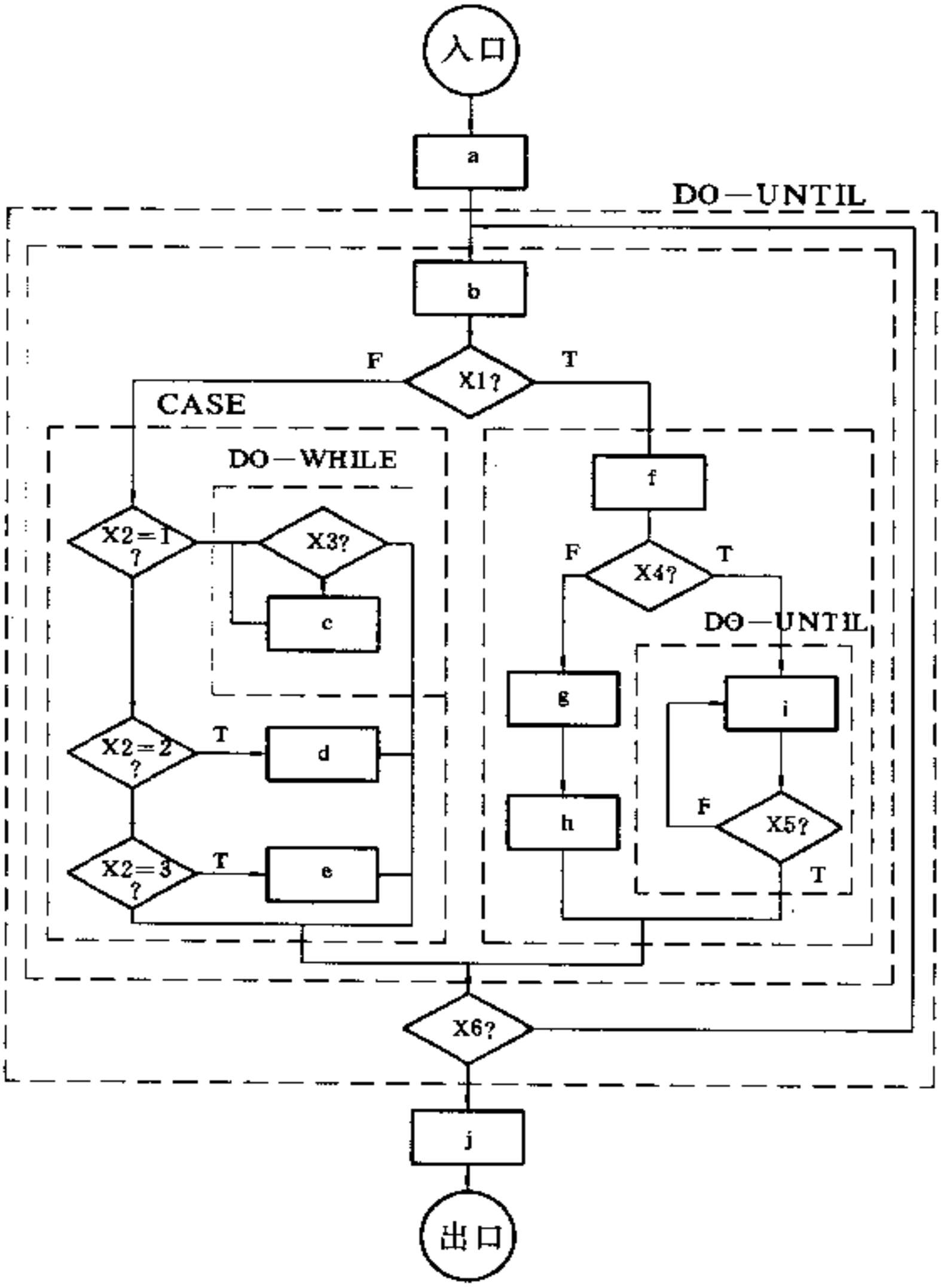


图 4.2　嵌套构成的流程图实例

出一个程序的流程图。图中增加了一些虚线构成的框，目的是便于理解控制结构的嵌套关系。例如，X_3 和 C 外面的虚线表示了它们构成了 while 型循环。但这一虚线框又是其外层虚线所包围的一个 CASE 型结构中控制变量 X_2 取值为 1 时要执行的部分。显然，这个流程图所描述的程序是结构化的。

其次，需要对流程图所使用的符号作出明确的规定。除去按规定使用定义了的符号外，流程图中不允许出现任何其它符号。在此特别向读者推荐近年为国际标准化组织提出，并已为我国国家标准局批准的一些程序流程图标准符号。图 4.3 给出这一标准的 14 种流程图符号，其中多数符号所规定的使用方法和普通的习惯用法一致。需要说明的几点是：

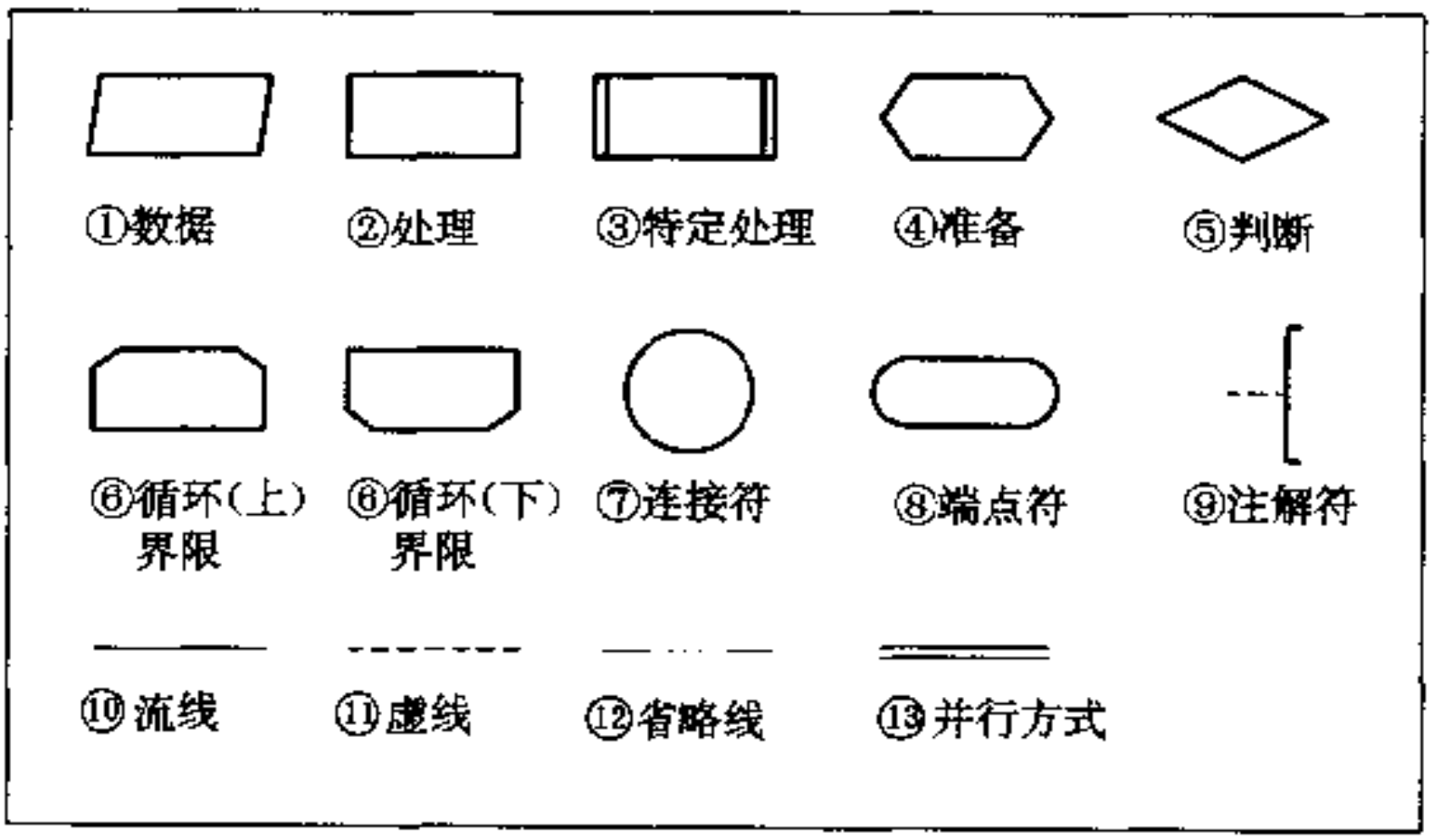


图 4.3　标准程序流程图的规定符号

(1) 循环的界限设有一对特殊的符号。循环开始符是矩形削去了上面两个直角，其中应注明循环名和进入循环的条件(while 型循环)，循环结束符是矩型削去了下面的两个直角，其中应注明循环名及循环终止的条件(Until 型循环)。通常这两个符号应在

一条纵线上，上下对应，其间夹有要重复执行的循环体。参看图4.4表示的两种类型循环的符号使用方法。

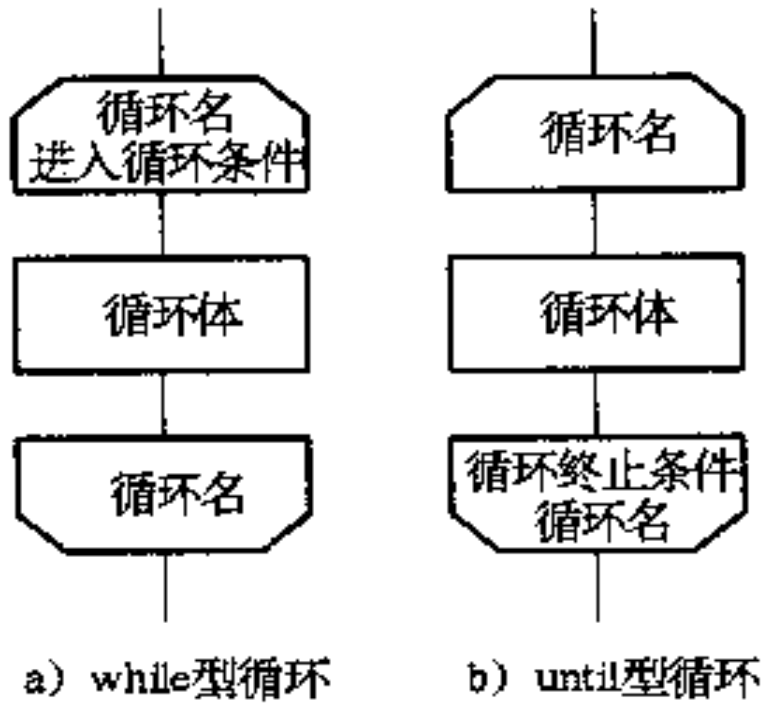


图 4.4　循环的标准符号

(2) 判断有一个入口，但有多个可选出口。在判断条件取值后有一个且仅有一个出口被激活。取值结果可在流线附近注明。显然，两出口的判断就是前面提到的选择型结构，多出口的判断即为 CASE 型结构。图 4.5 给出多出口判断的表示。其中(a)、(b)

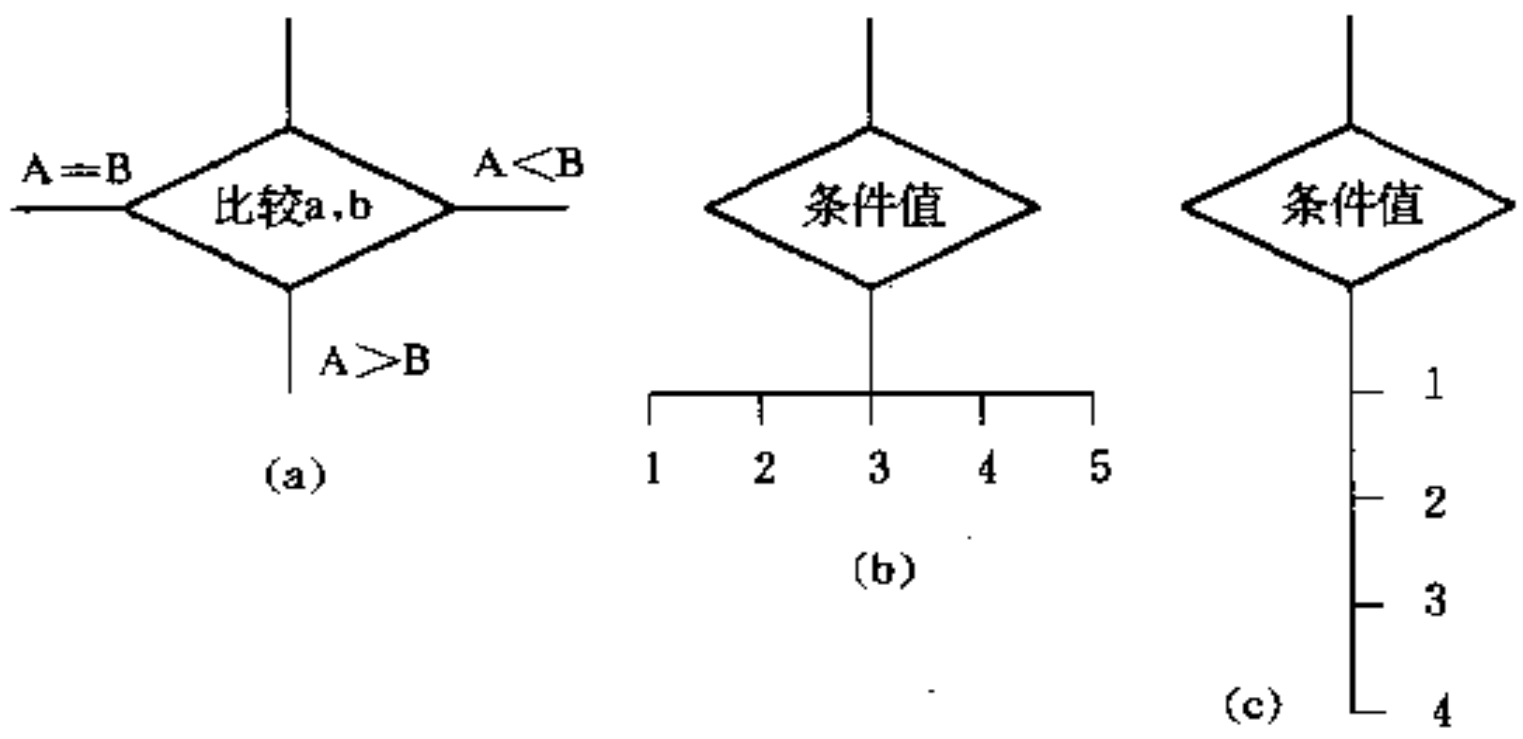


图 4.5　多出口判断

和(c)图分别表示具有 3、5 和 4 个出口判断。

(3) 流线表示控制流的流向，除特别加上表示方向的箭头者外，通常指的流向是自上而下和自左到右。

(4) 注解符可用来标识注解内容，其虚线联接在相关的符号上，或连接一个虚线框(框住一组符号)。参看图 4.6 的例子。

(5) 虚线表示两个或多个符号间的选择关系(例如，虚线连接了两个符号，则表示两符号只选用其中的一个)。虚线也可配合注解使用，也请参看图 4.6 中的举例。

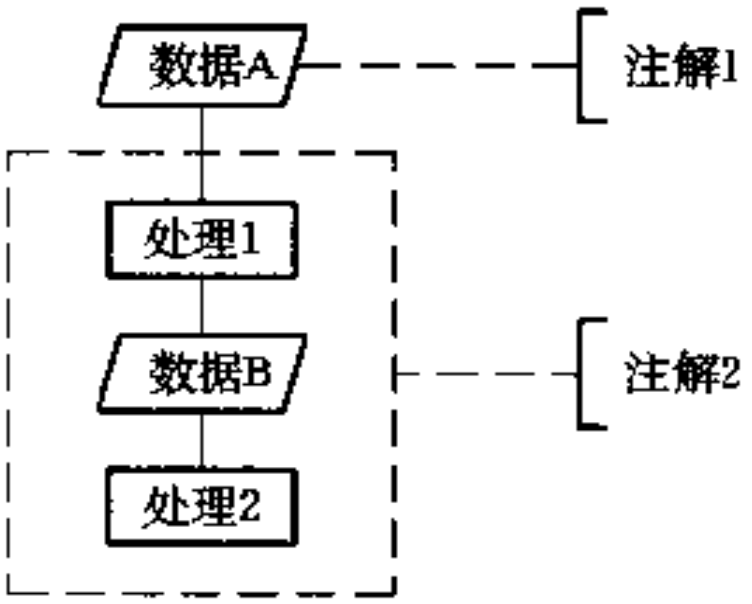


图 4.6　注解符的使用

(6) 端点表示转向外部环境或从外部环境转入。

图 4.7 是按这里介绍的标准流程图规定画出的图 4.2 表示的程序。请注意图中的循环表示和控制流的流向。把它与图 4.2 对比，程序结构的层次就显得更清楚了。

4.2　N-S 图

N-S 图的名称取自其创造者 Nassi 和 Shneiderman 两人名字的第一个字母。人们也称它为盒状图(Box diagram)。实际上，N-S 图是流程图的一个变种。为了较彻底地解决程序结构化的问题，N-S 图完全去掉了流程图中的控制流流线和箭头。因而完全排除了因任意使用控制转移对程序质量的影响。

对应于前述五种基本控制结构，N-S 图提供了五种图形构件(见图 4.8)。其中(a)图表示按顺序执行 A，再执行 B。(b)表示若条件取真值，执行“T”下面框 A 的内容；取假值时，则执行“F”下面的框 B 的内容。(c)和(d)表示的是两种类型的循环，P 是循环条件，S 是循环体。这两个符号的形状和它们的含意(一个是先判

断 P 的取值，再执行 S，另一个先执行 S，再判断 P 的值)多少有些联系，比较容易被人们接受。(e)给出了多出口判断的图形表示，P 为控制条件，根据 P 的取值，相应地执行其值下面各框的内容。

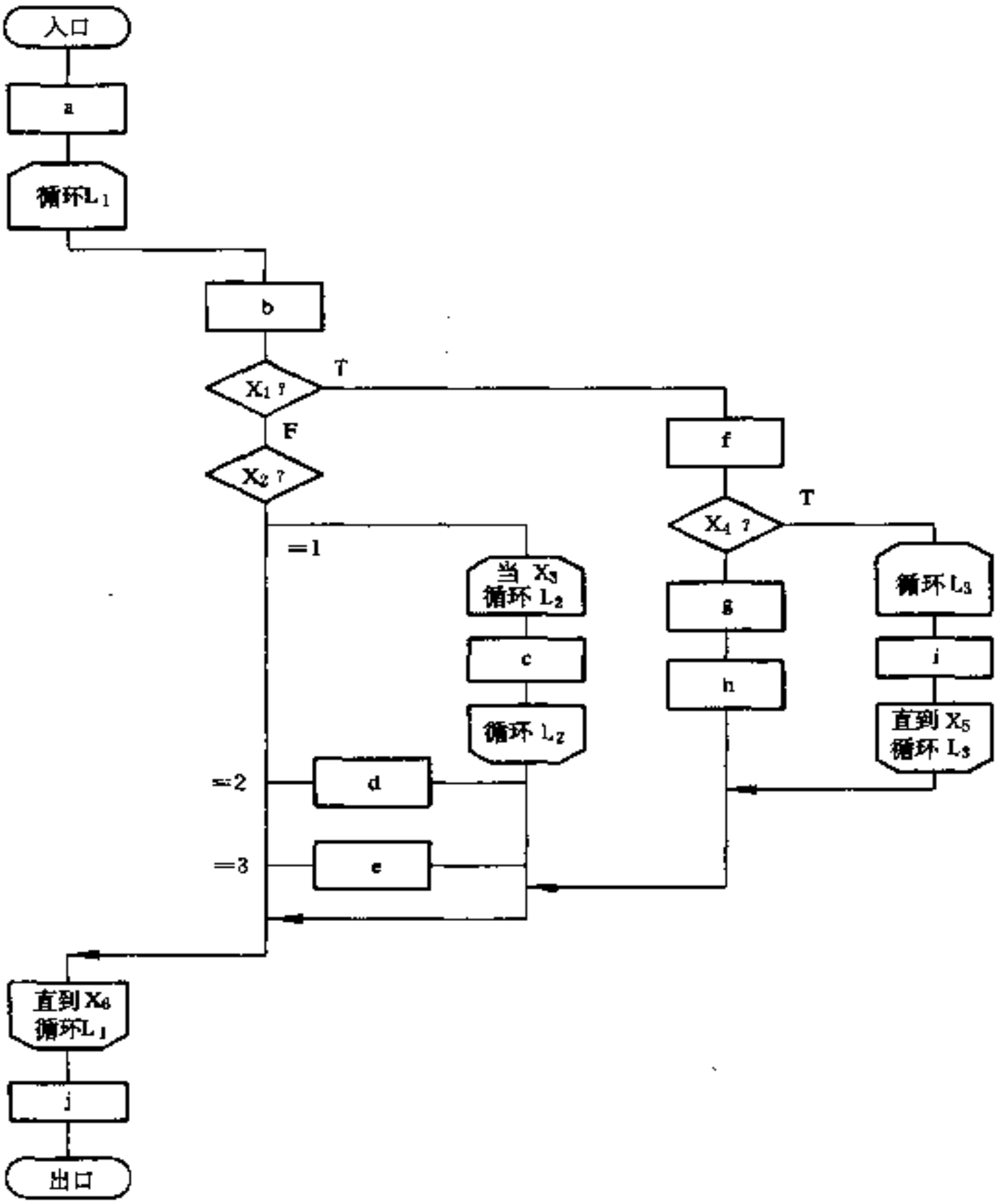
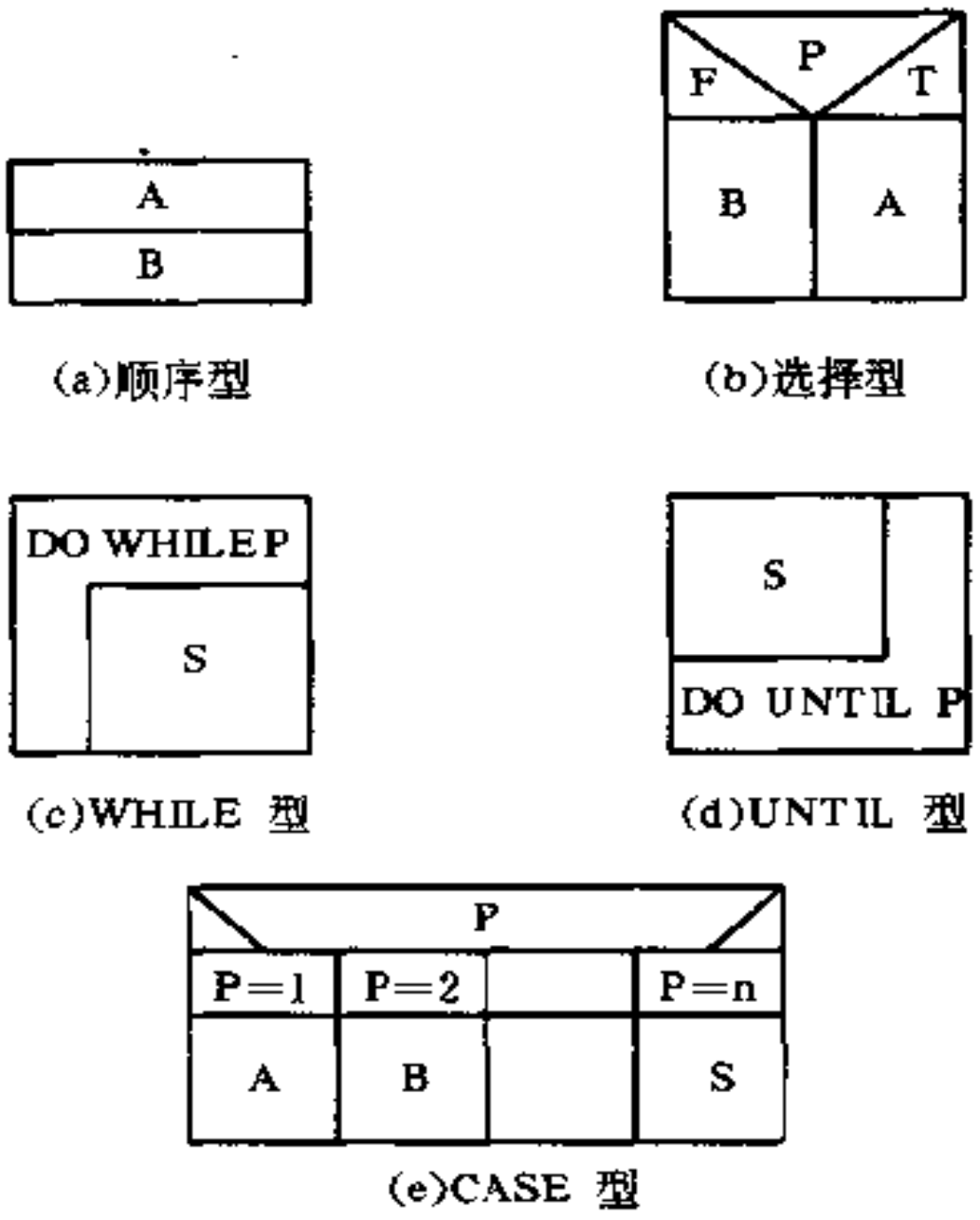


图 4.7　标准流程图的应用实例



(a)顺序型　　(b)选择型

(c)WHILE 型　　(d)UNTIL 型

(e)CASE 型

图 4.8　N—S 图的基本控制结构

为了说明 N-S 图的使用，这里仍用前面流程图的实例，将图 4.2 的程序用 N-S 图表示，如图 4.9 所示。

请注意 N-S 图的层次嵌套特性。从图 4.9 中可以看出，N-S 图表达的程序随着层次的增加，逐步从外向内进入图的核心部分。

图中每一个矩形(除 CASE 构造中条件取值矩形，如图 4.9 的 $X_2=1, X_2=2, X_2=3$ 外)都是明确定义了的功能域，也就是程序的执行段。如图 4.9 中的 a、b、c、d、e、f、g、h、i 和 j 各矩形框。

显然，利用 N-S 图设计的程序必然是结构化程序。另一方面，非结构化程序用 N-S 图是无法表达的。

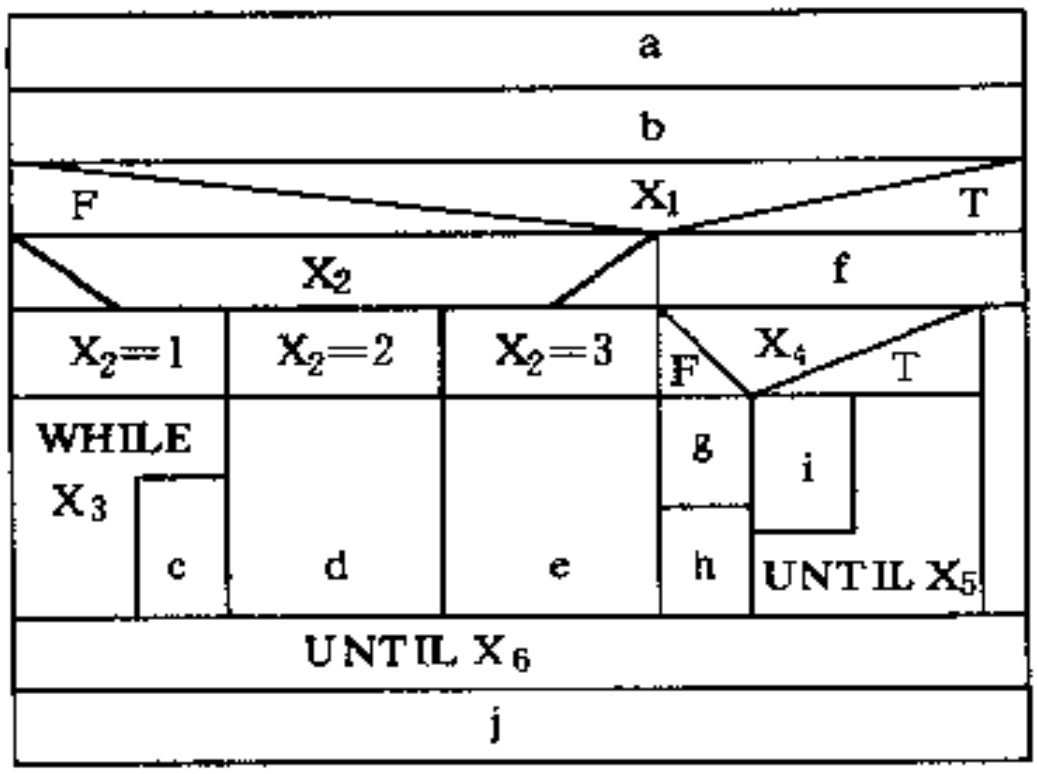
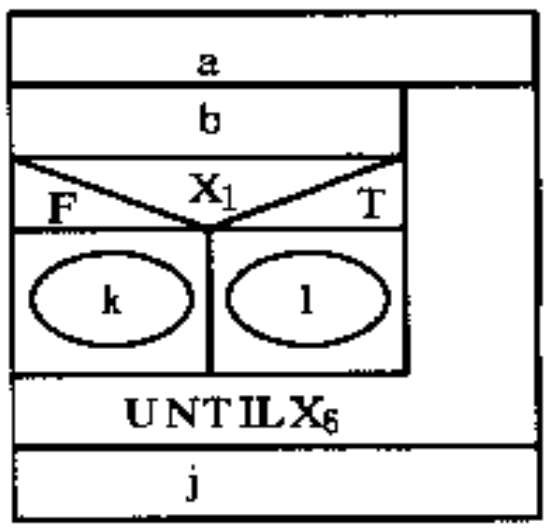
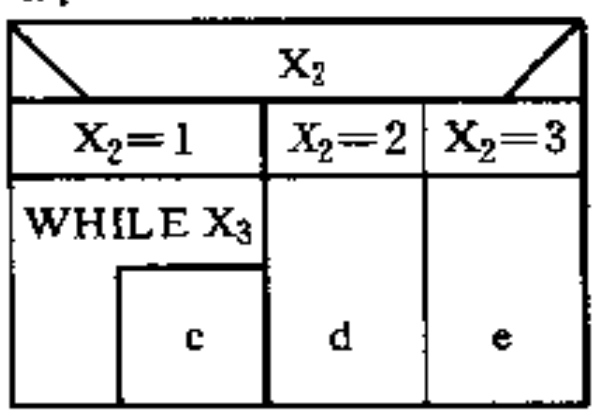


图 4.9　N—S 图实例



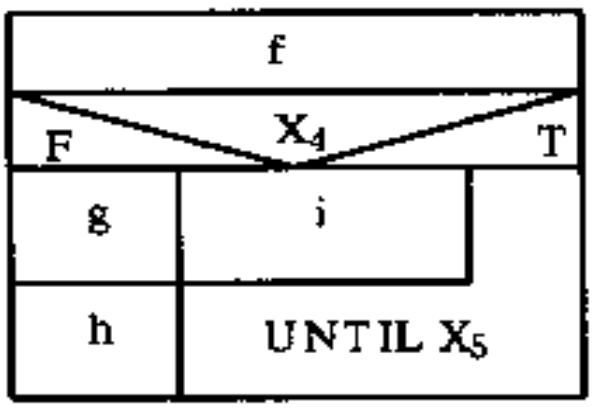
(a)

k：



(b)

l：



(c)

图 4.10　N—S 图的扩展表示

使用 N-S 图进行程序设计时，从图形的外层结构开始，逐步向内层扩展。但很可能对图的整个布局考虑不周，致使为内层开辟的矩形功能域过小，而不便于向内层继续扩展。遇到这种情况，一个补救的办法是在某个矩形功能域中给出椭圆形标记，其中标有符号，然后另行开辟新的矩形功能域。例如，图 4.10 中的 k 和 1 两功能域便是另辟出的。

4.3 PAD 图

PAD 是 Problem Analysis Diagram 的缩写，是几年前由日本日立公司提出，其后得到一定程度的推广。近年来国内也有一些单位采用作为详细设计的图形工具。

PAD 图也是从流程图演化而来的，它针对流程图的某些缺点，进行了适当的改进。把程序控制流结构表示成二维树的图形，与原来的流程图相比有着明显的优点。

PAD 图提供的五种基本控制结构如图 4.11 所示。其中(a)表示按顺序先执行 A，再执行 B；(b)给出了判断条件为 P 的选择型结构。P 取真值时执行上面的 A 框；P 取假值执行下面 B 框的内容。若这种选择结构只有上面的 A 框，而没有 B 框，其含义自然是

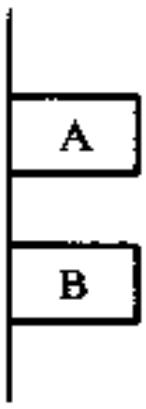
```
IF      P      THEN      A
```

(c)和(d)中 P 是循环控制条件，S 是循环体。注意循环条件右边矩形的双纵线，这是为了表明该矩形中给出的是循环条件，以区别于一般的矩形功能域。(e)给出 CASE 型结构，当表达式 P 取值为 1 时，执行 A 框的内容；取值为 2 时，执行 B 框的内容，等等。

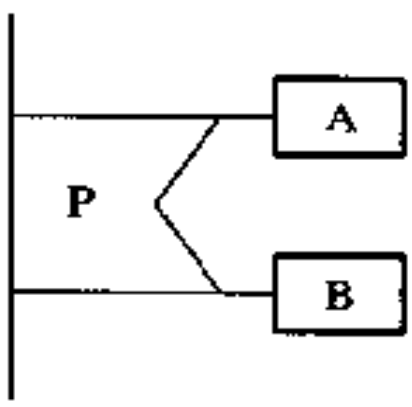
作为应用 PAD 图的实例，图 4.12 给出图 4.2 程序的 PAD 图。

PAD 图的优点有以下几个方面：

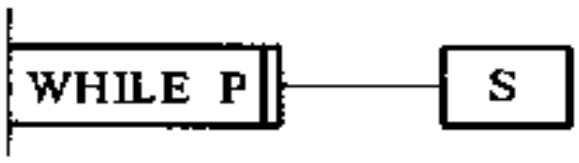
(1) 使用 PAD 图设计出的程序必定是结构化程序。



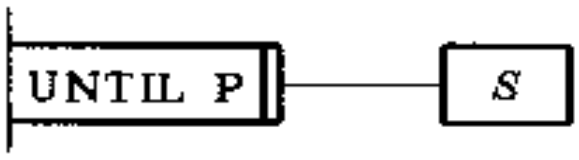
(a) 顺序型



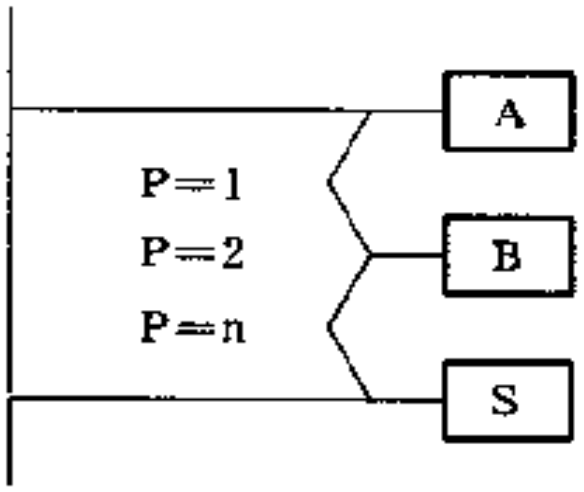
(b) 选择型



(c) WHILE型



(d) UNTIL型



(e) CASE型

图 4.11　PAD 图的基本控制结构

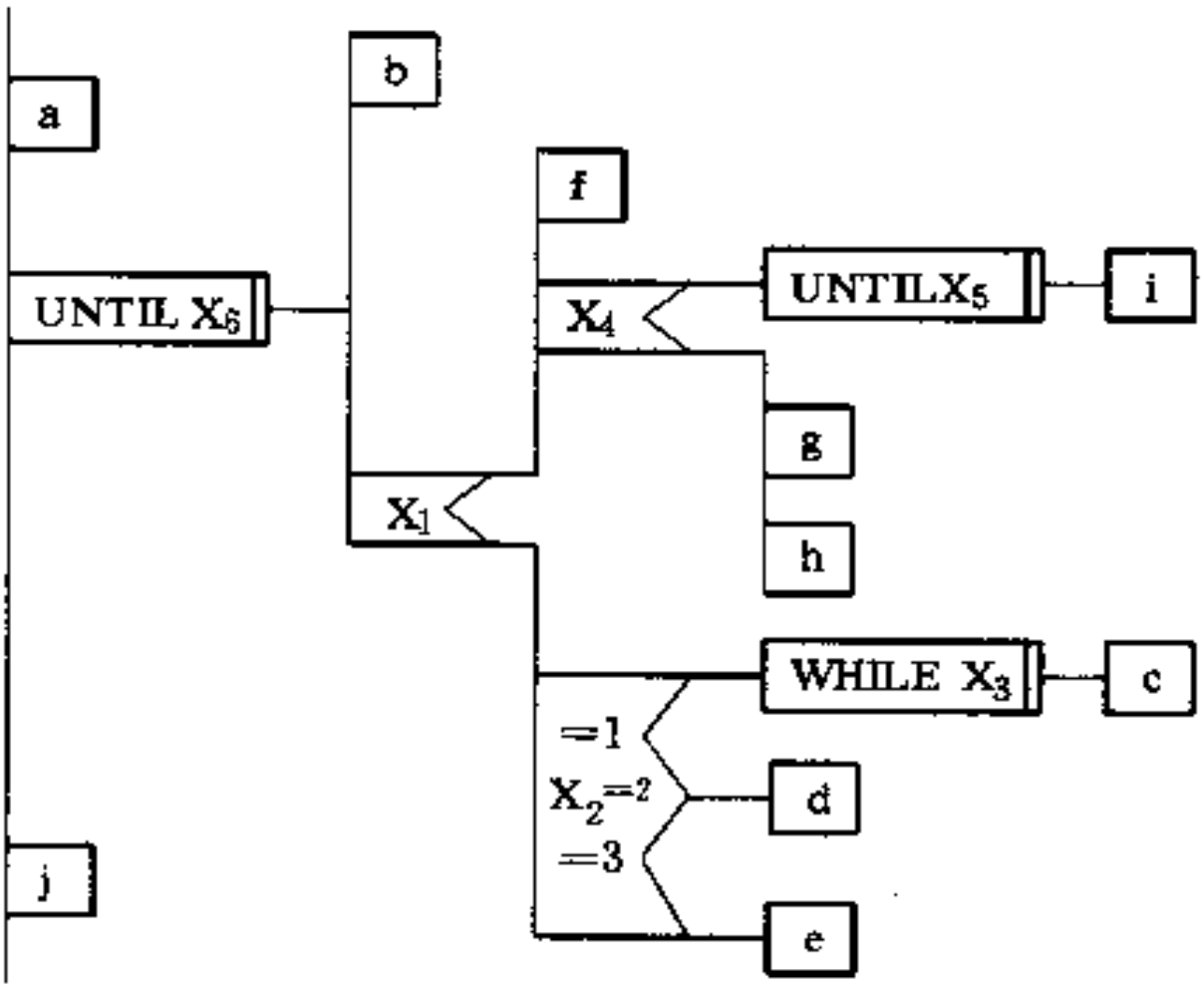


图 4.12　PAD 图实例

(2) PAD 图所表达的程序，其结构十分清晰。图中最左 纵线是程序的主干线，也是程序的第一层结构，随着层次的增加，图形逐渐向右扩展。每增加一个层次，图形向右扩展一道纵线。因此，程序中含有的层次数即为 PAD 图中的纵线数。

(3) 作为一种详细设计的图形工具，PAD 图比流程图更容易读。可以把每个 PAD 图看作一个树形结构，程序的执行顺序为从最左主干线上端的结点开始，自上而下，自左而右，以至遍历所有结点。

(4) 由于 PAD 图的树形特点，使我们比流程图更容易在计算机上进行处理。例如，在开发 PAD 图向高级语言程序的转换程序以后，便可从终端输入 PAD 图，并自动转换成高级语言程序。因而可以省去人工编码的步骤，从而大大地提高了软件开发的效率。

4.4 PDL 语言

近十年来出现了多种软件设计描述语言。这些语言形式上不同于上述的几种图形表示方法，却同样能在详细设计阶段，用以表达程序的逻辑结构。为区别于编写代码使用的高级语言，或程序设计语言(Programming Language)，我们称它们为设计程序用语言(Program Design Language)，简称 PDL。也有人把 PDL 称作伪代码或结构化语言。

PDL 语言所描述的程序是什么样子？让我们先来看一个实例——查找错拼单词的程序。为方便阅读，这里把结构化英语和结构化汉语(其中的关键字仍保留)同时给出，以利对照：

```
Procedure SPELLCHECK is
begin
   split document into single words
```

```
look up words in dictionary
display words which are not in dictionary
creat a new dictionary
end SPELLCHECK
```

```
Procedure 查找错拼单词 is
begin
  把整个文件分离成单词
  查字典
  显示字典中查不到的单词
  造一新字典
end 查找错拼单词
```

---

(1) PDL 程序的主要特征

从上面的实例中我们可以看出 PDL 程序的大部分特征：

① 用 PDL 写出的程序具有正文格式，在计算机上可作正文处理。这一点和高级语言写出的程序很相似。

② PDL 程序中含有一些能够标明程序结构的关键字。这些关键字常常以加了底线的英文字的形 式 书 写。例如，procedure，begin，end，loop，if，then，else，exit when 等。这种表示和印刷出的黑体英文字 **procedure，begin，end，loop，if，then，else，exit，when** 是没有区别的。但有时也用大写英文字表示关键字。

③ PDL 语言仅有少量的简单语法规则，大量地使用了人们最习惯的自然语言语句。这就为灵活方便地描述程序算法以及提高可读性创造了良好的条件。为描述程序的算法，软件人员把精力集中在算法的逻辑上，而不必过分受到语法规定的限制。

④ 使用 PDL 语言，常常按逐步细化的方式写出程序。从比较概括和抽象的 PDL 程序起，逐步写出更为精确、细致、接近于高级语言程序的 PDL 程序来。

前面给出的例子看起来比较粗糙，为进一步表明查找拼错单词的四个步骤是怎样实现的，可给出细化的 PDL 程序(参看图 4.13 及图 4.14)。

```
procedure SPELLCHECK is
begin
    —— * split document into words
    loop
    get next word
    add word to word list in sort order
    exit when all words processed
end loop
—— * look up words in dictionary
loop
 get word from word list
 if word not in dictionary then
    —— * display words not in dictionary display word, prompt on user ter-
    minal
    if user response says word OK then add word to good word list
    else
            add word to bad word list
    end if
 end if
    exit when all words processed
    end loop
    —— * create a new dictionary
    DICTIONARY: = merge dictionary and good word list
end   SPELLCHECK
```

图 4.13

```
procedure 查找错拼单词 is
begin
    ——*把整个文件分离成单词
    loop
        取下一单词
        按排序顺序把单词插入单词表
        exit when 所有单词均已处理完
    end loop
    ——*查字典
    loop
        从单词表中取单词
        if 单词不在字典内then
            ——*显示未在字典中查到的单词在用户终端上立即显示单词
            if 用户回答说此单词有意义then 把单词列入正常单词表
            else
              把单词列入异常单词表
            end if
      end if
      exit when 所有单词均已处理完
end loop
——*造一新字典
字典:=把字典与正常单词表合并
end 查找错拼单词
```

图4.14

⑤ PDL 程序的注释行对语句进行解释，起到提高可读性的作用。图4.13和图4.14中以

——*

为开始的行均为注释行。

(2) PDL 程序的构成

和许多高级语言类似，完整的 PDL 程序包含有数据说明、子程序结构、分程序结构、顺序控制结构以及输入/输出结构。以下分别加以说明。

① 数据说明

数据说明在 PDL 程序中用以指明数据名的类型及作用域，其形式为

declare〈数据名〉as〈限定词〉

其中，〈数据名〉是被指明的变量或常量的名字表，

〈限定词〉是具体的数据结构，如：

scalar（纯量）

array（数组）

list（列表）

char（字符）

structure（结构）

② 子程序结构

procedure〈子程序名〉

interface〈参数表〉

〈分程序和(或)PDL 语句〉

return

end〈子程序名〉

其中，〈PDL 语句〉是指各种 PDL 构造。

③ 分程序结构

begin〈分程序名〉

〈PDL 语句〉

end〈分程序名〉

④ 顺序控制结构

1)选择型

if〈条件〉then

〈PDL 语句〉

else

〈PDL 语句〉

end if

其中，〈条件〉是可取真假值的 PDL 语句。当有多个条件复合时，可用elseif 结构，如：

if〈条件〉then

〈PDL 语句〉

elseif〈条件〉then

〈PDL 语句〉

else

〈PDL 语句〉

endif

这两种选择型控制结构的应用实例是：

```
•if sensor—value is in range then
      display sensor—value
else
     display alarm—message
end if
```

```
•if current—character is backslash then
     turn on indenting
elseif
     last character is period then
     turn off indenting
else
     print current—character
```

endif

2)WHILE 型循环

loop while〈条件〉

〈PDL 语句〉

end loop

在〈条件〉取真值时，执行循环体，否则结束循环。

3)UNTIL 型循环

loop until〈条件〉

〈PDL 语句〉

end loop

在〈条件〉取真值时退出循环体，和它等价的形式是：

loop

〈PDL 语句〉

exit when〈条件〉

end loop

这两类循环的例子是：

•loop while there are more transaction records in inpur file

read next transaction record from input file

verify transaction—record

queue transaction—record

end loop

•loop until data is last day(of this month)

read inventory record from stored inventory file

get date(month, day, year)

write inventory record on output file

end loop

4) CASE 型

case〈选择因子〉of

〈标号〉{，〈标号〉}：〈PDL 语句〉

… …

[default]　　：[〈PDL 语句〉]

end case

其中，〈选择因子〉是具有一组已知值的数据元素，这些值即是其下面各行的〈标号〉。case 结构将根据〈选择因子〉的取值，执行相应行上〈标号〉所指的 PDL 语句。花括号表明，标号可以有多个，相邻标号之间应有逗号。方括号内的语法元素是任选的，可有可无。[default]指出了〈选择因子〉可能选取的最后一种情况。

以下给出两个 case 型结构的实例：

```
case month—name  of
  February          : check for leap year
                      process February
  January, March, May,
  August, October,
  December          : process 31 day month
  April, June, September,
  November          : process 30 day month
end case
case today of
Monday              :compute initial balance
  Tuesday, Wednesday,
  Thursday          :generate daily updates
  Friday            :compute closing balance
  default           :do nothing
  end case
```

⑤ 输入/输出结构

PDL 应具有输入和输出功能，但输入和输出结构的具体形式

常常根据情况作出特殊的约定，一般采用print，read，display 等语句是很常见的。

(3) PDL 和高级语言的差别

从上面介绍的 PDL 程序构成及实例中，我们会发现它和高级语言有许多相似之处。但它毕竟不是一种高级语言，必须注意它们之间的差别。

① 在概要设计完成以后，软件人员需要把注意力集中到模块的功能如何实现方面，但在考虑实现的算法时，又不会被实现一些过分细小的问题所缠绕。他需要一种有力的概括表达工具，PDL 能在详细设计阶段适应这一要求。另一方面，高级语言则完全是为编码服务的，它难于完成上述要求。因此，两者适用的开发阶段是完全不同的。

② 通常高级程序语言都有一套严格细致的语法规则，计算机不能接受违反了语法规则的程序。PDL 语言的语法规则便没有那么严格，并且它允许使用自然语言的语句。因此，使用起来灵活方便得多，也最容易使用和被别人看懂。“结构化英语”只是其中使用了英语，“结构化汉语”则是其中使用了汉语的句子。如果把 PDL 语言称为程序人员的本国语言和计算机算法语言之间的桥梁并不算过份。

③ PDL 程序不能直接进行编译，但由于它具有和高级语言相似的程序构造和正文形式，能够很方便地在计算机上进行存放、处理和输入输出，如果得到特定的软件支持，可以把 PDL 程序自动地转换成高级语言程序，从而大大地提高了软件开发的效率。

④ 遵循自顶向下、逐步细化的原则。采用 PDL 语言写出的程序可以很抽象、很概括、很简单，例如，PDL 程序中可能有：

“计算每个工作人员的月工资”

这样的语句。在逐步细化以后，PDL 程序中也可能出现非常细致

而具体的细节，接近甚至相当于高级语言的语句，例如：

Temporaty = (average(i, n+1)−sum)/average(i,i)

把细化以后的 PDL 程序转换成高级语言程序已经是轻而易举的工作了。从这个意义上理解，我们完全可以说，PDL 程序是从高级逻辑抽象过渡到详细算法描述的桥梁。

4.5 HIPO 图

HIPO 是 Hierarchy plus Input-Process-Output 的缩写。它是1976年由 IBM 公司提出的，一开始只是用作文件编写的格式要求，随后发展成为比较有名的软件设计手段。

(1) HIPO 图的构成

构成 HIPO 实际上只是两个部分：目录表和 IPO 图。目录表给出的是程序结构的层次关系，IPO 图则为程序各部分提供具体的工作细节。现分别简介如下：

目录表 VTOC 是 Visual Table of Contents 的缩写，也称为 H 图。它是反映程序功能结构的树状图，和本书前一章概要设计中介绍的程序结构图非常相似。树结构的各结点也是能完成特定功能的模块，每个模块均标以简要的功能或模块名字(参看图4.15)。请注意各模块中右下角的编号，有了编号可以更方便地找到各层模块间的关系，甚至可联系到源程序对应和程序段，我们称此为具有可跟踪性。图中右下角附有图例，可对各模块给出必要的说明。所作说明均为自然语言的简明语句。如有必要，图例还可能对图中所用符号给出定义，以方便阅读。通过 VTOC 可以了解程序功能的概貌。在许多项目中，它和程序的结构是一致的。VTOC 中的模块如何划分以及该图的宽度、深度如何考虑等问题都可参考第三章中有关程序结构图的处理办法，恰当地掌握。

IPO 图则为每一功能模块详细地指明要有哪些输入、完成哪

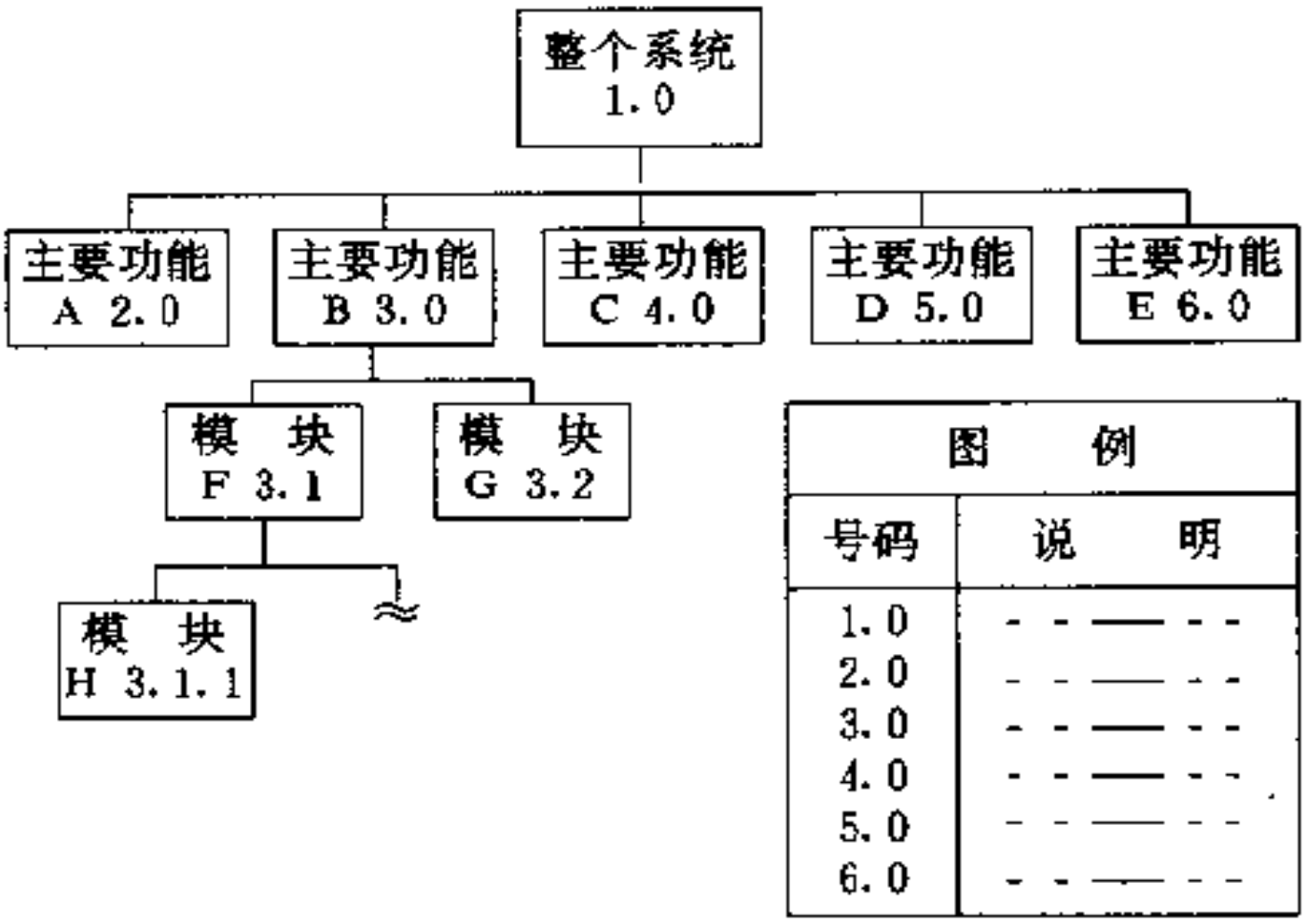


图 4.15 HIPO的VTOC

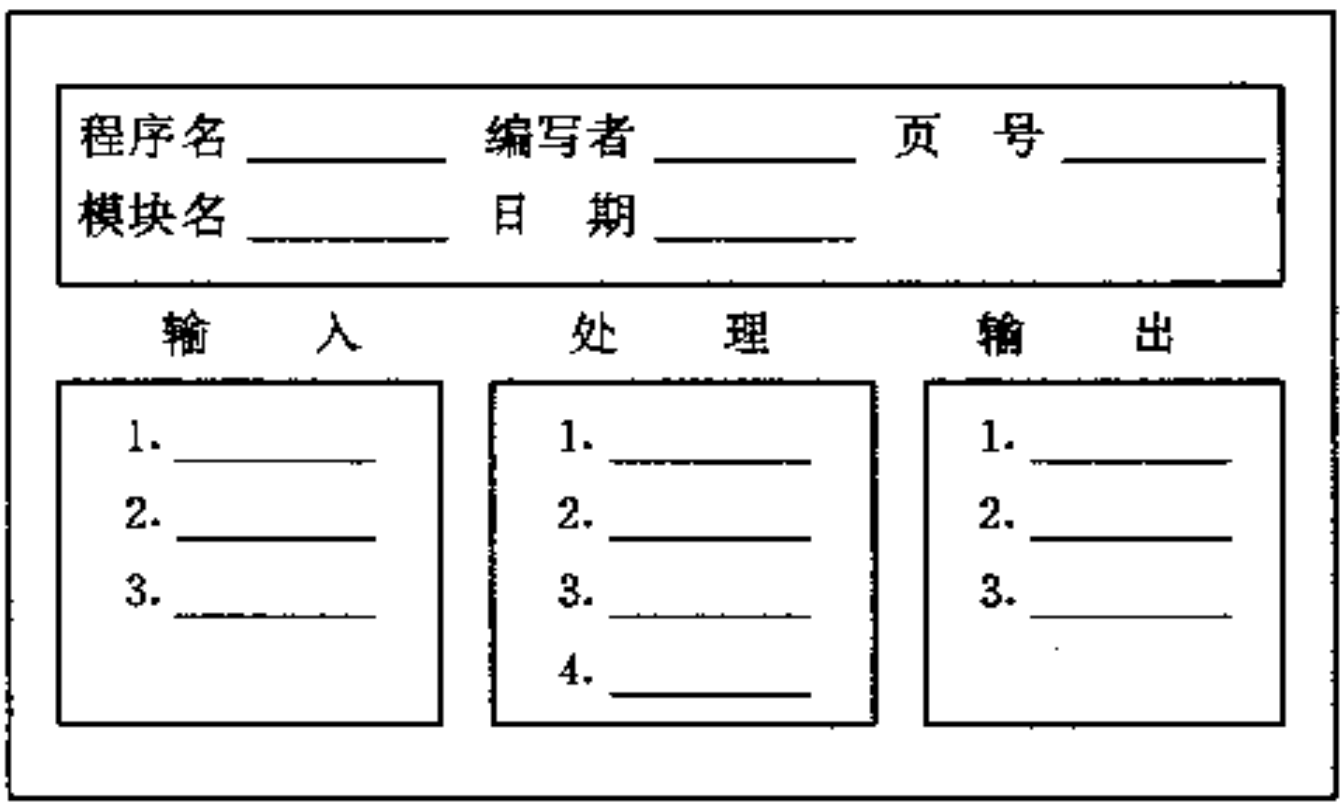


图 4.16 IPO 图

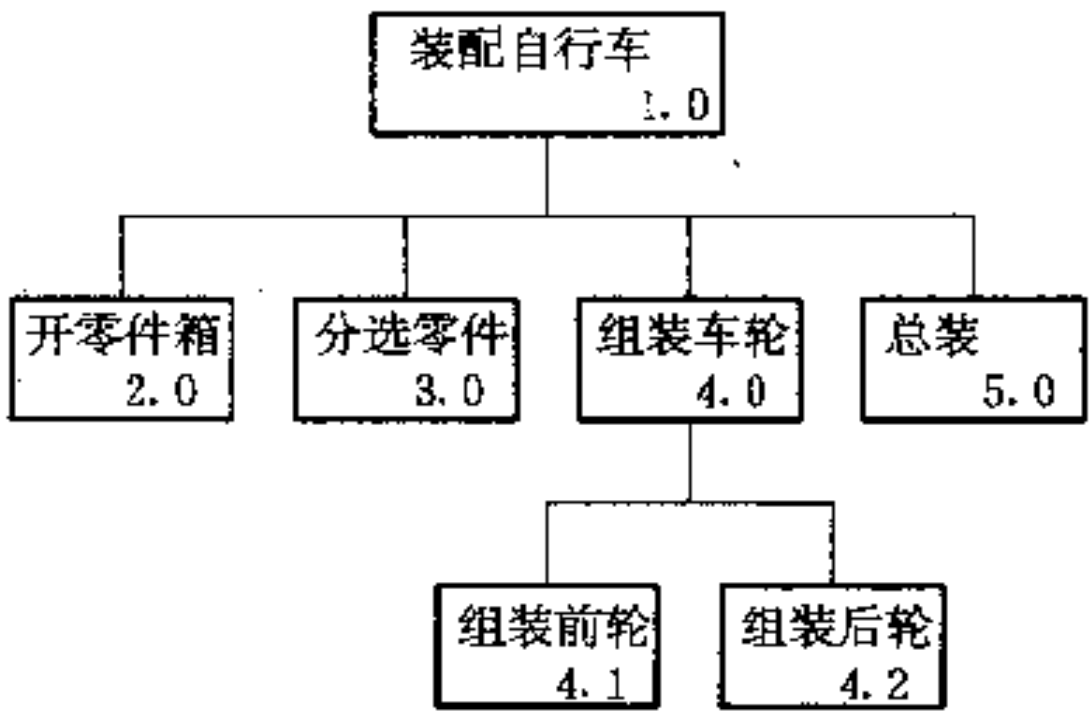
些操作以及得到哪些输出数据(参看图4.16)。通常 IPO 图有固定的格式，图中操作部分总是列在中间，输入和输出分别在其左右。由于某些模块的细节很难在一张 IPO 图中表达清楚，常常把

IPO 图又分成两段，简单概括一些的 IPO 图称之为概要图（overview diagram），细致具体一些的称之为细节图（detail diagram）。设计 IPO 图的关键问题是把中间的处理部分描述清楚。通常使用的是非常简要的自然语言语句，其中主要应包括为了描述某个操作是什么的动作。但应注意，动词选用得不当，比如过分抽象是不能说明问题的。表4.1列出 IBM 公司推荐使用的动词，可供选用。

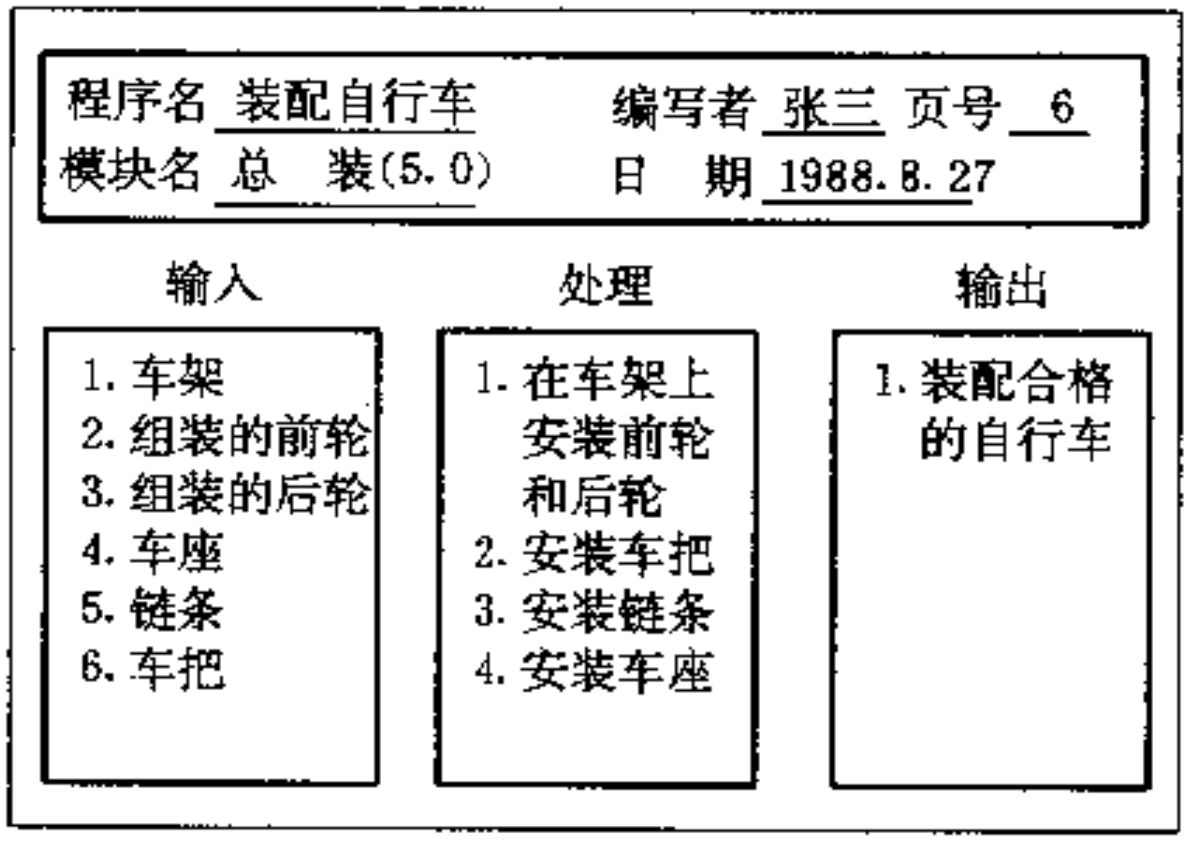
表 4.1　HIPO 所用动词

accept	delete	get	perform	select
add	dequeue	handle	place	set
allocate	detach	identify	position	specify
alter	determine	increment	post	start
analyze	display	initialize	process	stop
assign	do until	insert	provide	store
begin	do while	issue	purge	supply
build	edit	locate	put	suspend
calculate	encode	link	queue	switch
check	enqueire	load	read	terminate
clear	enter	look up	record	test
close	establish	maintain	reinstate	transfer
complete	examine	make	release	translate
construct	execute	merge	resolve	update
control	exit	modify	restore	use
convert	extract	monitor	return	validate
copy	find	move	scan	verify
create	fix	obtain	schedule	write
decrement	format	open	search	

图4.17给出了描述自行车装配的 HIPO 图。



(a)装配自行车的VTOC图



(b)装配自行车的总装IPO图

图4.17 装配自行车的 HIPO 图

(2) 利用 HIPO 图进行迭代式细化设计

HIPO 图作为一种设计描述工具，它应该服从于设计工作的需要，适应设计工作的要求。在软件开发过程中，软件人员解决设

计问题常常要经历思维逐步发展的过程，由粗到细，由部分到全体，并且对一些问题还要经过反复的考虑才可能达到比较满意的设计效果。我们称此为迭代式细化设计(iterative refinement design)。HIPO 能很好地适应这一要求，图4.18是利用 HIPO 进行迭代式细化设计的示意图。该图表明，把 VTOC 和 IPO 结合起来，反复交替地使用它们，使得设计工作逐步深化，最终取得完满的设计结果。其实这正是自顶向下，逐步求精的结构化程序设计思想。关于结构化程序设计，我们在下一章还要仔细讨论。

(3) HIPO 图的特点

比起其它设计表达工具来，HIPO 有着自己的特点。首先，这一图形表示方法非常容易为人们看懂。用 HIPO 描述的设计，即使没有计算机专业知识的人也很容易接受。除去图形具有直观效果以外，这还因为其中使用了自然语言的语句。对于软件高级人员、初级人员、管理人员以及用户来说都认为它是交流设计思想的好形式。

其次，HIPO 的适用范围很广，绝不限于详细设计。事实上，画出 VTOC 图就是和概要设计密切相关的工作。如果利用它仅仅表达软件要达到的功能，则是需求分析中描述需求的很好工具。由于 HIPO 图是在开发过程中的表达工具，它又是软件开发的文档编制工具。开发完成 HIPO 图就是很好的文档，而不必在设计完成以后，专门补写文档。

此外，上面介绍的 HIPO 图画法仅仅是最基本的要求，实际的软件开发项目若采用它时，还可根据具体情况作些规定或扩充，以适应项目开发的需要。因此，使用 HIPO 有一定的灵活性。例如，在 IPO 图中的处理部分，如果不用编号条文形式，而是使用前面介绍的 PDL 语句，也会取得很好的使用效果。最后，使用 HIPO 图开发软件还可便于人员的分工，以及有利于测试、纠错等优点。

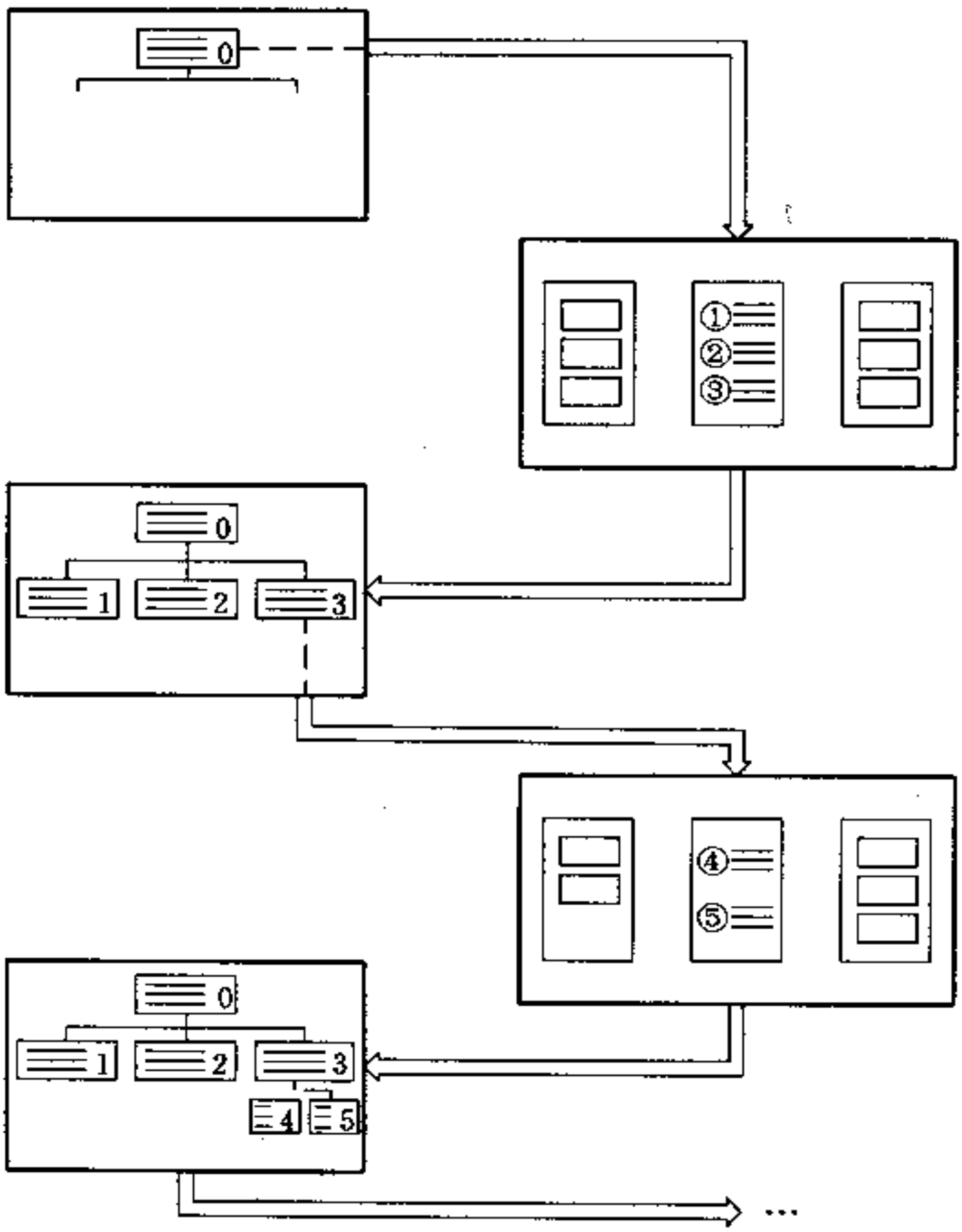


图 4.18

4.6 选用详细设计表达工具的原则

以上介绍的详细设计表达工具，如何选择使用呢？这很难用一句话回答，除习惯等主观因素外，还有：①易于学习掌握；②易于表达逻辑条件及其相应的处理；③易于进行逻辑验证；④便于转换成机器能接受的代码；⑤便于机器读入及处理；⑥便于修改；⑦体现了结构化的要求；⑧目前软件人员使用的普遍性；⑨能有效地表达各种数据类型和数据结构。

第五章　结构化程序设计与程序设计风格

软件开发工作沿着生存期的顺序，逐步推进，最终要得到能在计算机上运行的程序，我们通常称它为源程序。把软件设计作进一步转换，产生源程序的过程，称为编码阶段。实际上，源程序中体现了前面各个开发阶段软件人员所付出的大量劳动。按照软件工程的原则，软件项目的开发更重视前面各阶段的工作。这些工作已经为编码打下了良好的基础，主要的困难在需求分析和设计阶段已经解决。编码工作只是把详细设计得到的算法描述转换成某种语言表示的程序，相对而言，编码要比前面几个阶段的工作容易得多，普通的程序员完全可以胜任编码工作。这一点和早期关于"程序设计"，或者"程序编写"的传统认识有着很大区别。

为了保证编码的质量，程序员必须深刻地理解、熟练地掌握并正确地运用程序设计语言的特性，例如一些语法规则和语义的细节。只有语法上没有错误的程序才能通过编译系统的语法检查。然而，软件工程项目对代码编写的要求，绝不仅仅是源程序语法的正确性，也不只是源程序中没有各种错误(请参看第六章6.1节软件测试的目的)，它还要求源程序具有良好的结构性和良好的程序设计风格(programming style)。本章将针对这两方面的要求展开讨论，其目的是使读者了解到，在软件开发项目中，并不是源程序写得正确就完全达到要求了。

5.1　对源程序的质量要求

为什么程序的正确性不是对程序质量的唯一要求呢?

也许有一天计算机有能力去理解人们用自然语言描述的程序要求，那将极大地简化了软件开发工作。或者能做到，这台计算机能为那台计算机编写源程序，甚至一台计算机能为自己编写程序。如果真的能达到这一步，软件开发和维护工作就方便多了，不仅可以免去软件设计和源程序编写的繁重而复杂的脑力劳动，而且可以免去令人厌烦的阅读程序之苦。然而，这毕竟是未来若干年后才能做到的事。目前和今后若干年内，人们编写源程序还只能用某种程序设计语言，并且写出的源程序除送入计算机运行外，还必须让人能够容易看懂。这一点作为软件工程项目和软件产品是一个必不可少的质量要求。如果某个软件开发项目，写好的源程序在其它方面都十分完美，只是不容易被人们看懂，那还不是一个好的程序。实践表明，一个软件产品完成开发工作，投入运行以后，如果发生了问题，很难依靠原来的开发人员来解决。因为人员的工作调动是不可避免的，即使找到了当时的程序编写者本人。他们大多已无法记起几年前编写程序的许多细节了。无论如何，程序编写时，就应考虑到，所写出的程序将被别人阅读，一定要尽量使程序写得容易被人读懂。60 年代初，人们当时估计开发出的程序最多使用五年。可事实上，在 IBM 4361 上运行的程序有些是 25 年前编写的，现在还在使用，并且不知道还要使用多久。很显然，无论怎样，也不可能让开发者本人维护它 25 年。尽管对于不同性质的软件在这方面可能存在某些要求的差别，然而改善或提高程序的可读性总是程序人员为之奋斗的一个目标。

假如我们写出的源程序便于阅读，又便于测试和排除发现的程序差错，就极大地减小了运行中发生错误的可能性。使得已开发的软件在经过测试和调试以后，消除了大多数隐藏的差错，因而可以做到正常稳定地运行。在指定的时间内不发生故障的概率得到提高，即改善或提高了软件的可靠性。

假如我们写出的源程序在运行过程中发现了问题或错误，很容易修改，并且能在运行的环境中，根据用户的需要比较容易地扩充其功能，改善其性能，那么这样的程序，其可维护性(maintainability)较好。维护人员可以很方便地对它完成修改、扩充和完善。

本章下面的讨论着重从程序的结构化和程序设计风格方面，如何保证达到上述程序的质量要求。

5.2 结构化程序设计

结构化程序设计是 60 年代末至 70 年代中形成的技术，它主要包括两个方面：

①对代码编写时使用控制结构的要求，强调使用几种基本控制结构，避免使用可能降低程序结构性的转向语句。

②在软件开发的设计与实现过程中，提倡采用自顶向下(Top—down)和逐步细化(Stepwise refinement)的原则。

(1)结构化程序

早在 1963 年 ALGOL 60 语言报告文本的主要作者 Peter Naur 提出，在程序中由于大量地、无限制地使用 GO TO 语句，使得程序结构非常混乱。许多转向语句把程序的各个部分勾联在一起，如同一堆乱麻。事实上，自从计算机问世以来的前 25 年中，人们写出的大多数程序正是纷乱如麻的(参看图 5.1)。

1965 年 E. W. Dijkstra 提出，由于 GO TO 语句造成程序结构的混乱，致使程序质量下降。为了解决这一问题，应该把 GO TO 语句从高级语言中取消。已经广泛流行的高级语言 FORTRAN，COBOL 和 ALGOL 暂且保留。新开发的语言中则不应再有 GO TO 语句。例如，纯 LISP、BLISS、ISWIM 等语言正是这样做的。

1966 年 Böhm 和 Jacopini 证明了任何单入口和单出口，且没有“死循环”的程序都能由三种最基本的控制结构构造出来。三种

基本控制结构是：

顺序结构

选择结构——IF…THEN…ELSE…

循环结构——DO WHILE 或 DO UNTIL

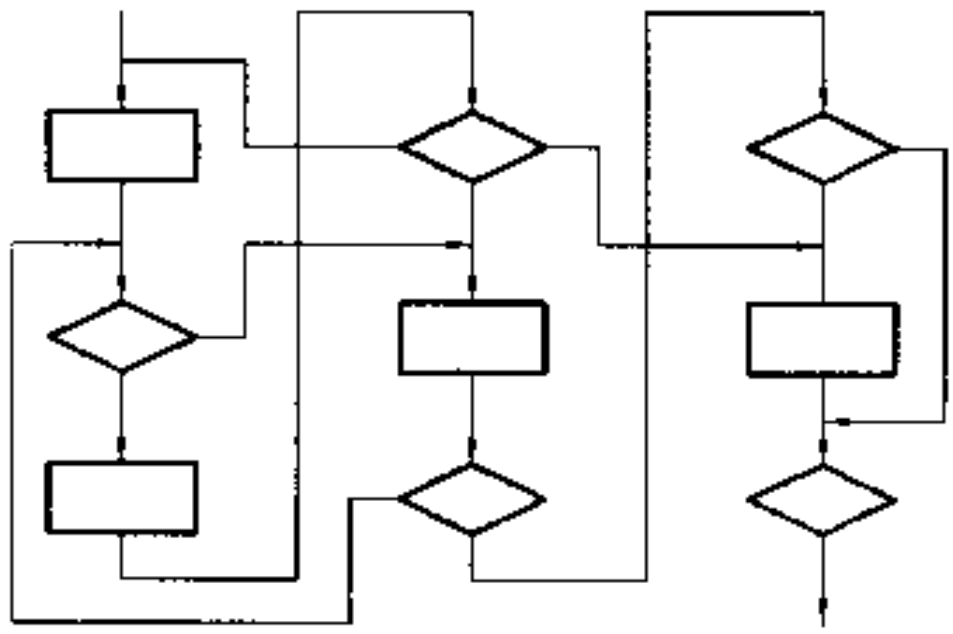


图 5.1　纷乱如麻和程序结构

这样构成的程序可以完全避免使用 GO TO 语句。紧接着展开了一场十分激烈的争论，争论集中在如何看待 GO TO 语句的问题上，赞成延用 GO TO 语句的人认为，使用 GO TO 语句可在程序编写中得到很大的灵活性，并且 GO TO 语句可直接得到硬件 jump 指令的支持。经过争论大家明确了，问 题不只是涉及到要不要 GO TO 语句。实际上，争论的实质是对程序应该提出什么样的质量要求，怎样才能得到高质量的程序。这就不能不涉及到程序设计的思想和方法。

70 年代初 N. Wirth 在设计 Pascal 语言时对 GO TO 语句的处理可被当作对 GO TO 语句争论的结论。在 Pascal 语言中设有支持上述三种基本控制构造的语句；另一方面，GO TO 语句仍然保留在该语言中。不过 N. Wirth 解释说，通常使用所提供的几种控制语句已经足够了，习惯于这样做的人不会感到 GO TO 语句的必要。也就是说，一般情况下，可以完全不用 GO TO 语句。如果在

特殊情况下，由于特定的要求，偶然使用 GO TO 语句能解决问题，那也未尝不可，只是不应大量使用它罢了。

(2) 结构化语言

早期的程序设计语言，由于没有考虑到程序结构化的要求，自然没有直接支持几种基本控制结构的特性。比如，FORTRAN、COBOL、ALGOL 和 BASIC 语言中大都没有一些与基本控制结构相对应的控制语句。我们称这些语言为非结构化程序设计语言。

对于那些考虑到结构化要求的新语言，如 Pascal、C 等，由于具有与基本控制结构对应的控制语句，可以直接编写出结构化程序来，我们称之为结构化程序设计语言。虽然在这些语言中也包含了 GO TO 语句，但一般编写程序时并不使用。图 5.2 给出了 Pascal 和 FORTRAN Ⅳ控制语句的比较。

如何使用非结构化程序设计语言进行结构化程序设计是值得我们重视的问题。虽然有了新的语言，但仍然不能把老的语言全都作废，那是因为原有的硬件、软件、任务和人们的习惯等诸因素决定的。但是今天我们使用这些语言绝不能再写出“繁乱如麻”的程序来。

以下举 FORTRAN Ⅳ语言为例，用它来进行结构化程序设计，也就是利用 FORTRAN Ⅳ语句建立几种基本控制结构。

使用 FORTRAN Ⅳ语句建立选择型结构

if P then A else B

可以这样做：

```
      IF (.NOT.P) GO TO L1
       A
      GO TO L2
L1    CONTINUE
       B
L2    CONTINUE
```

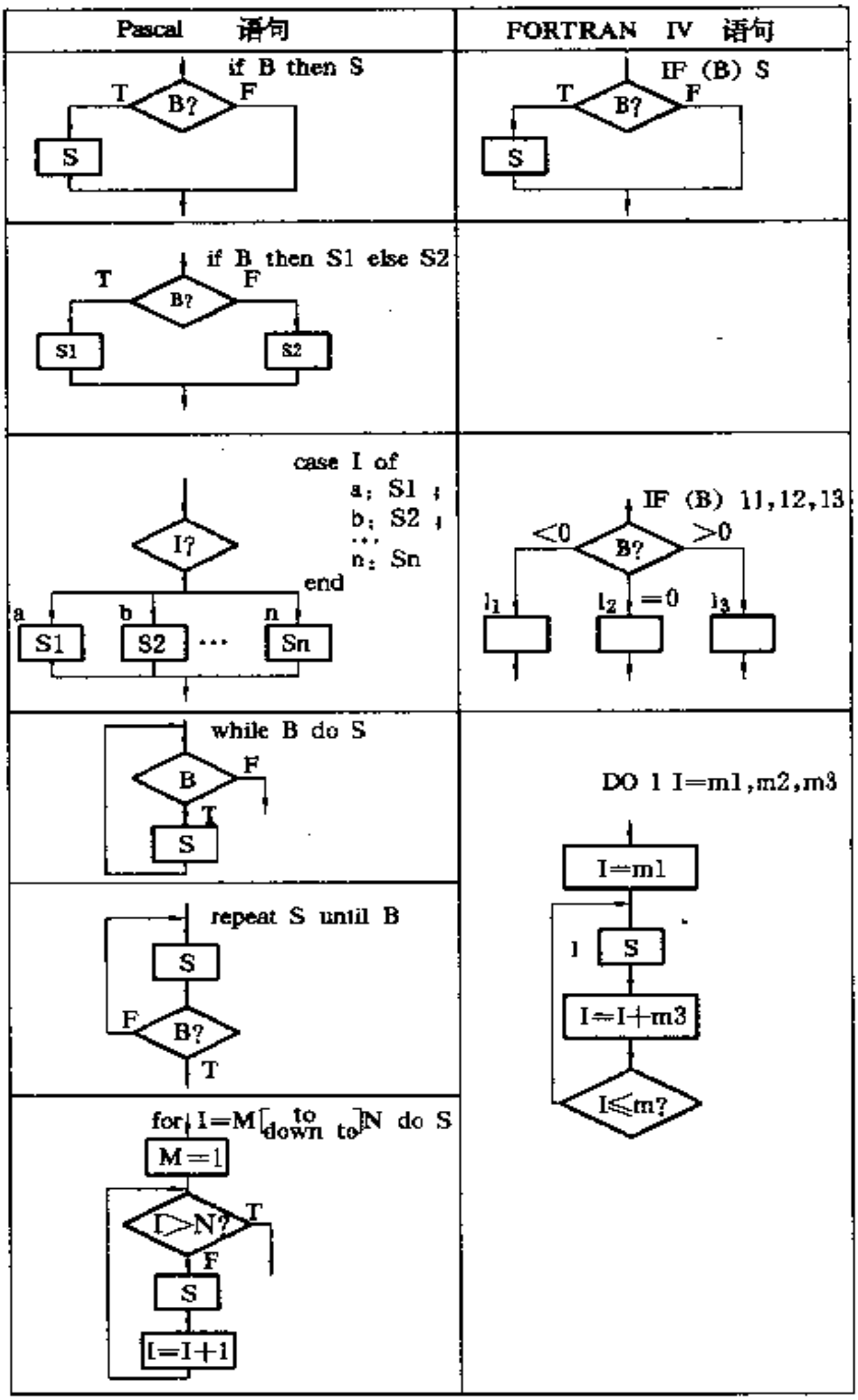


图 5.2 Pascal 和 FORTRAN IV 控制语句的对比

其中，第一个语句是FORTRAN Ⅳ的逻辑条件语句，L1和L2是语句标号。也可以不用否定逻辑运算符 .NOT.：

```
        IF (P) GO TO L1
               GO TO L2
C THEN
  L1 A
     GO TO L3
C ELSE
  L2 B
     GO TO L3
L3 CONTINUE
```

其中，左端为C的是注释行，L1，L2，L3是语句标号。

使用 FORTRAN Ⅳ语句建立循环结构

while P do S

实现的程序段为：

```
L2 CONTINUE
   IF(.NOT. P) GO TO L1
   S
   GO TO L2
L1 CONTINUE
```

使用 FORTRAN Ⅳ语句建立循环语句

do S until P

实现的程序段为

```
L1 CONTINUE
   S
   IF(.NOT. P) GO TO L1
   CONTINUE
```

可以使用 FORTRAN Ⅳ的计算转向语句来实现多出口选择结构

Case P of (A; B; …; S)

```
      K=P
      GO TO(L1,L2,…,LN), K
L1    CONTINUE
      A
      GO TO LM
L2    CONTINUE
      B
      GO TO LM
      ⋮
LN    CONTINUE
      S
LM    CONTINUE
```

(3) 自顶向下与逐步细化

自顶向下与逐步细化是结构化程序设计的原则，它把整个设计过程分出层次来，逐步加以解决。每一步是在前一步的基础上进行的，是前一步设计的细化和具体化。这可以从一个树结构图上来理解(参看图 5.3)。树结构上每一层都有一些要解决的问题，这些问题往往是相互独立的。各层节点表示各个步骤，在每一节点上都要解决怎样进一步分解，怎样细化的问题，以便得到在它下面的各个子女节点。这种做法就把一个原来复杂的大问题，划分成多个容易解决的小问题，最后整个问题得到逐步解决。

按照自顶向上、逐步细化的原则进行设计有着许多好处，比如：

① 同一层节点上的细化工作彼此独立无关。

② 任何一步发生了错误，只能影响到它所在树枝的子女节点。

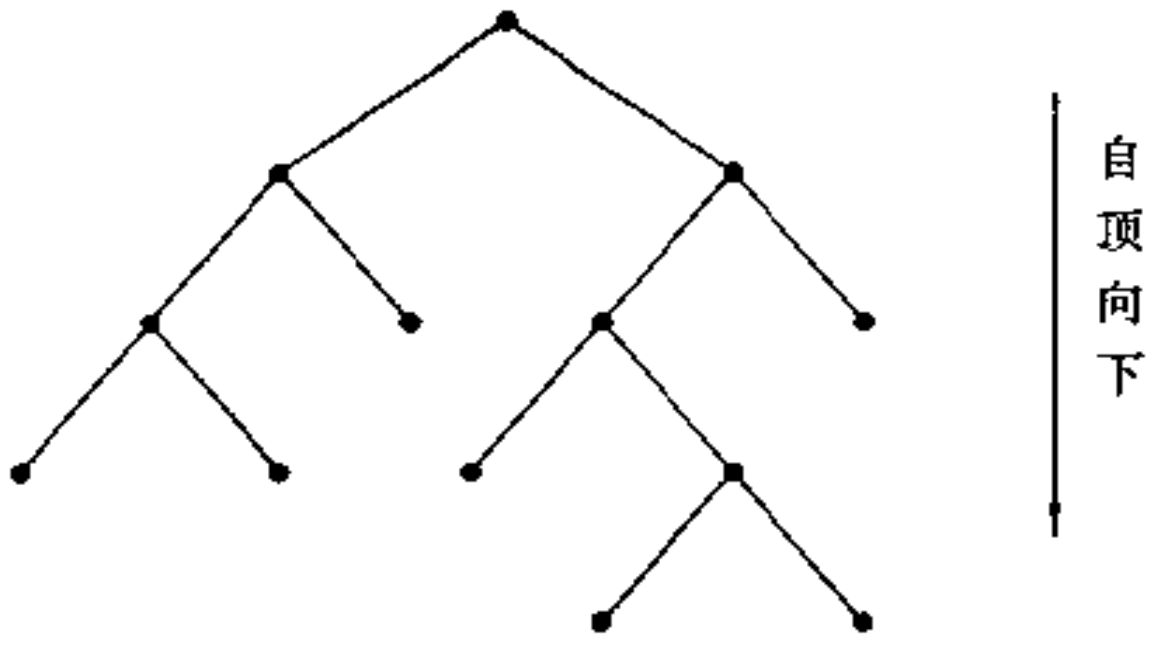


图 5.3　自顶向下，逐步细化的树结构

③ 测试工作可按顺序，逐个节点独立进行，最后再集成。

④ 每一步工作仅在上层节点基础上作不多的设计扩展，便于检查。

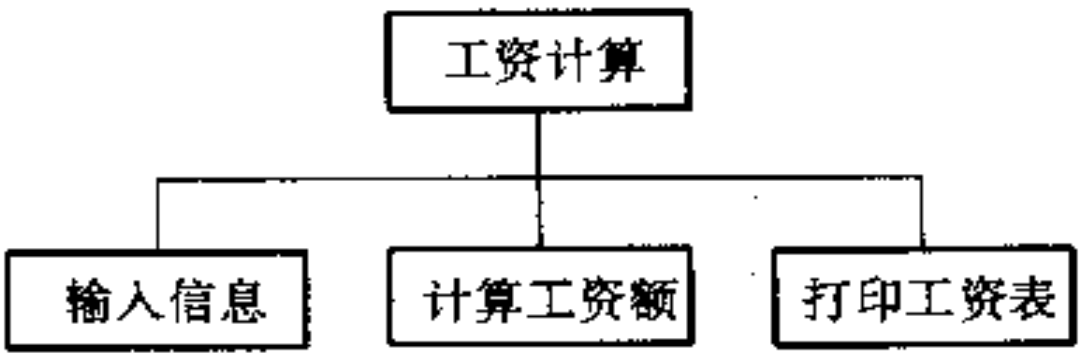
⑤ 有利于设计的分工和组织工作。

在此以工资计算的程序为例，说明上述的设计原则。在采用自顶向下进行设计时，可把整个工资计算程序当作顶层的树根节点。它的工作可分成三部分. 即读入工资数据、计算工资额和打印出工资表。于是可用图 5.4(a)来表示。进一步要着重解决工资的计算问题，通常实发工资和应发工资与扣除额有关，而应发额又与基本工资和奖金额有关，等等。这样按图 5.4(b)直至分解到基层的叶节点为止，在这些节点上，问题已经很容易解决。

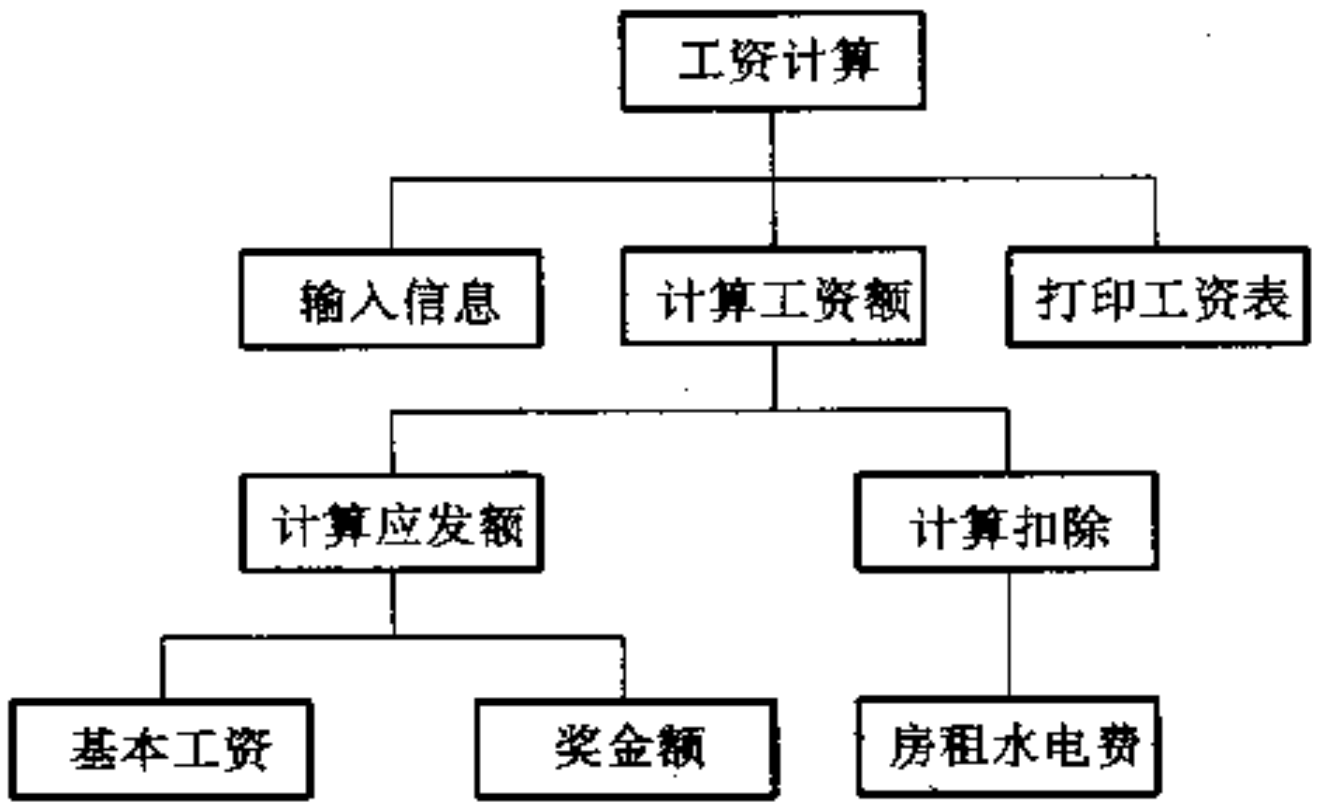
为进一步说明自顶向下，逐步细化的设计原则，这里再给出一个用 C 语言开发程序的实例。该例要求利用筛选法求 100 以内的质数。所说的筛选法，是从 2 至 100 中去掉 2，3，…，9，10 的倍数，剩下的便是 100 以内的质数。以下分为六步，逐步建立程序：

第一步　从 2—100 中筛弃 2—10 的倍数

第二步　建立 2—100 的表



（a）



（b）

5.4　工资计算程序的自顶向下开发

若 X 在表中，且 X 是 2—10 中任一数的倍数，则去掉 X

第三步　建立数组 A[100]，

其中某一元素 A[i]=i 若是 2—10 中任一数的倍数，则去掉 A[i]

第四步　建立数组 A[100]

其中某一元素 A[i]=i 若是 2，3，5，7 中任一数的倍数，则去掉 A[i]

第五步　for(i=2; i<=100; i+ +)

A[i]=i;

```
(对于 j=2, 3, 5, 7)
  for(i=2; i<=100; i++)
    if(A[i]/j*j==A[i])
      A[i]=0;
  for(i=2; i<100; i++)
    if(A[i]!=0)
      printf("A[%d])=%d/n", i, A[i]);
```

第六步

```
for(i=2; i<=100; i++)
  A[i]=i;
B[1]=2; B[2]=3; B[3]=5; B[4]=7;
for (j=1; j<4; j++)
  for(i=2; i<=100; i++)
    if(A[i]/B[j]*B[j]==A[i])
      A[i]=0;
for(i=2; i<=100; i++)
  if(A[i]!=0)
    printf("A[%d]=%d/n", i,A[i]);
```

(4) H. Mills 最近指出，结构化程序设计问题除去从控制结构入手、消除不适当使用的 GO TO 语句外，还有数据结构的不合理使用的情况。使控制结构合理化只是解决结构化程序设计问题的初步成果，下一步还应解决数据访问的合理化和规范化问题.

目前在许多程序中广泛地使用着数组和指针，对于这两种数据结构中元素的存取完全依赖于它们的下标值，实际上下标的取值是随机的。这一混乱存取的现象和使用 GO TO 语句一样对程序是十分有害的。H. Mills 建议用栈结构和队结构去代替数组和指针，使得数据的存取分别遵循后进先出(LIFO)和先进先出(FIFO)的规律合理而规范地进行。有人已经证明了在程序中用栈和队把数组和指针替换掉是完全可能的。H. Mills 说，我们期待着这

一论点尽快为人们普遍接受，但恐怕至少需要十年时间。

最后，在结束有关结构化程序设计的讨论时，举一个日常生活的例子对照，也许能说明一定问题。几年以前电视机和收音机的生产水平没有达到今天的高度时，许多人，特别是无线电爱好者纷纷购买元器件自己组装。但由于线路和工艺极不规范，甚至元器件本身都不符合质量要求，这样组装成的整机工作起来常常是不可靠的。今天，对于这样的整机不用说出钱买，就是无偿赠送，人们也不愿接受。理由很简单，这样的整机运行中的维修是一个大难题。今天几乎没有人再这样做了，谁都希望得到便于维修和正规生产的整机。对比之下，软件产品的情况不是也十分相似吗？

5.3 程序设计风格

前面已经说明提高程序的可读性是保证程序的质量的重要方面。然而，由于源程序描述的许多算法细节主要是提供给计算机使用的，对人来说，太不直观了。图 5.5 表明了软件开发中各阶段得到的成果性质对比。在需求分析阶段得到的需求规格说明书主要是规定所开发的软件应做些什么，通常并不涉及到怎么做的问题。随着开发工作的推进，经历了概要设计、详细设计和代码编写阶段，这些阶段得到的成果必定越来越多地反映出实现的细节，同时其中反映做什么的因素会越来越少。使用某种语言编写的源程序常常是很不直观的，我们如果不特别注意采取一些措施努力提高它的可读性，往往很难看懂在这些运算的背后隐藏着什么意图。因此，编写源程序时，保持良好的程序设计风格，就是为提高程序“透明性”的有力措施。它要求在源程序编写时为今后的读者提供更多有关需求和设计的信息，最终达到提高可读性的目的。

(1) 引例

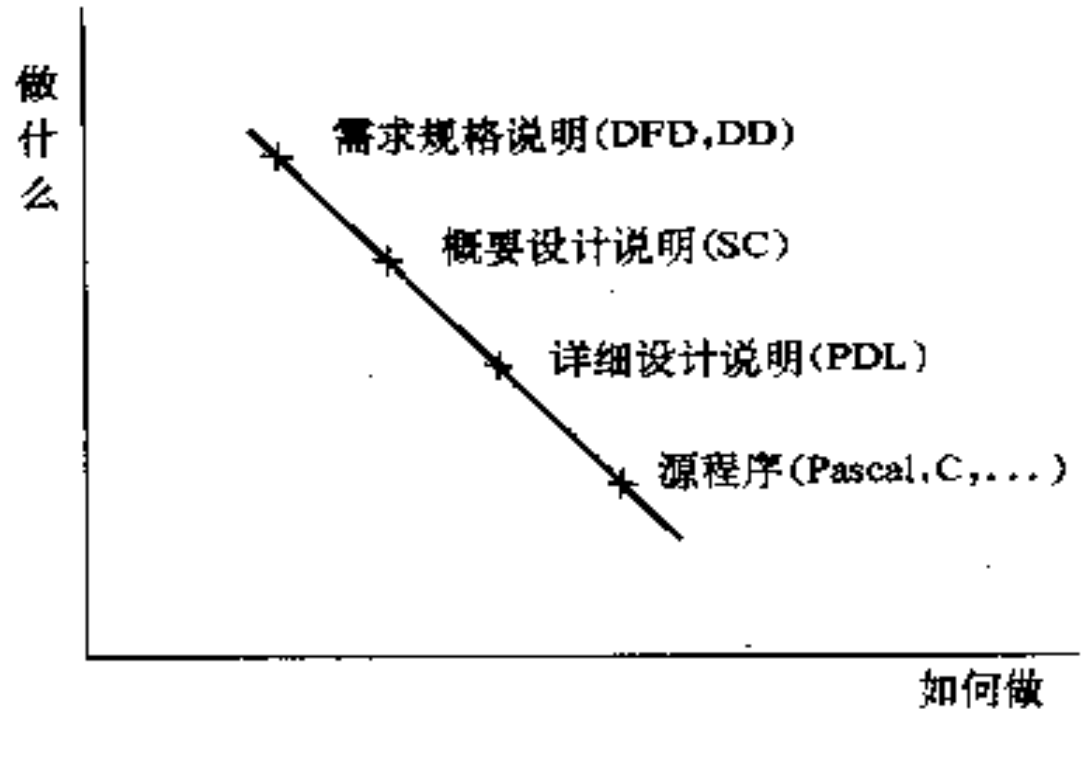


图 5.5 各开发阶段处理问题的侧重

什么是程序设计风格，让我们来看一个简单的例子。此例是由三个 FORTRAN 语句构成的程序段，试问读者从这三个语句中能够看出是在做什么吗？

```
      DO    5    I=1, N
      DO    5    J=1, N
5     V(I, J)=(I/J) * (J/I)
```

这是具有双重循环的程序段，得到的结果是一个 N×N 的二维数组，只是说明语句被略去了。我们知道，在 FORTRAN 语言中，除运算(/)在除数和被除数都是整型量时，其结果只取整数部分，而得到整型量。因此，

I/J 为 0　　当 I<J 时

J/I 为 0　　当 J<I 时

得到的数组 V(I, J)：

(I/J) * (J/I)为 0　　当 I≠J 时

(I/J) * (J/I)为 1　　当 I=J 时

这样得到的结果 V 是单位矩阵

$$\begin{pmatrix} 1 & & \\ & 1 & 0 \\ & \ddots & \\ & 0 & 1 \end{pmatrix}$$

实际上得到的结果还经过了整数向实数的转换。因此，最后结果单位矩阵 V 形为

$$\begin{pmatrix} 1.0 & & & 0.0 \\ & 1.0 & & \\ & & \ddots & \\ 0.0 & & & 1.0 \end{pmatrix}$$

这个结果恐怕很难一眼看得出来。阅读程序时人们要花很大力气弄清程序编制者的真正意图，这无疑给软件的维护带来很大困难。

上面的例子如果加以改写，只增加两行程序，阅读起来就要直观多了。

```
C   MAKE   V   AN   IDENTITY   MATRIX
      DO     5   I=1,N
      DO     3   J=1,N
3       V(I, J)=0.0
5       V(I,I)=1.0
```

(2) 构成程序设计风格的因素

要想得到具有良好风格的程序，应解决好以下几个方面的问题：

① 符号名的命名

在程序开发中经常要遇到为符号命名的问题，如变量、常量、标号、子程序、模块以及数据区、缓冲区等等。自然，如何命名有很大的任意性和灵活性，但我们不主张使用单个字母，或带数字的字母，如 A、B、C；A1、B2等。这不仅是因为这样可供使用

的名字数量有限，更主要的是无法体现名字所代表的符号的含意。最好采用一些具有实际意义的标识符，使其能够见名知意。例如，表明次数的量使用 TIMES，表明平均值的量使用 AVERAGE，表示总量的用 TOTAL，表示和的量用 SUM 等等。如果为标号命名，可使用 START、CALCULATE、INITIALIZE 等。在字符个数受限制的情况下，完全可以使用有意义的缩写。例如，用 AVERA 代表 AVERAGE，用 CALCU 代表 CALCULATE 等等。当然，如果必要也可使用汉语拼音命名。

② 程序中的注释行(comment)

许多程序设计语言允许使用注释行，使用它的目的完全是让读者更容易理解程序。我们应充分利用这一手段。认为写不写注释行无所谓的观点是错误的。它不是可有可无，而是一定要有。我们注意到，一些正规的程序文本，其中注释行的数量占整个程序总量的三分之一到一半，甚至更多。

怎样写注释得是个重要问题。以下分两方面加以说明。

任何一个程序或模块的开头都应有足够的序言性注释，它对于理解程序本身具有引导的作用。有些软件开发部门对序言性注释作了明确而严格的规定，要求程序编写者逐项列出。有关项目包括：

程序标题	引用的子程序
功能	对局部量的解释
调用形式	对局部量的解释
使用参数的意义	编者
输入数据	审查者
输出数据	日期

在没有这些注释的情况下阅读程序显然会有许多困难。

另一方面，夹在可执行语句中的注释行必须着重说明其后的语句或程序段是在进行什么操作，也就是要解释下面要“做什么”

或是执行了下面的语句会怎么样。而不要解释下面怎么做，因为解释怎么做常常是和程序本身重复的，并且对于阅读者理解程序没有什么帮助。例如，

```
/ * ADD AMOUNT TO TOTAL * /
TOTAL=AMOUNT+TOTAL;
```

这样的注释行仅仅重复了后面的语句，对于理解它的工作并没有什么作用。如果注明把月销售额计入年度总额，便使读者理解了下面程序的意图：

```
/ * ADD MONTHLY-SALES TO ANNUAL-TOTAL * /
TOTAL=AMOUNT+TOTAL;
```

③ 恰当地使用空格、空行和移行(indentation)

一个程序如果写得密密麻麻，分不出层次来常常是很难看懂的。优秀的程序员在利用空格、空行和移行的技巧上显示了他们的经验。恰当地利用空格，可以突出运算的优先性，避免发生运算的错误。例如，表达式

```
(A<-17)AND NOT(B<=49)ORC
```

写成

```
(A<-17)␣ AND␣ NOT(B<=49)␣ OR C
```

就更清楚。如果把表达式 D * A * * B 写成 D * ␣ A * * B 就无论如何不会理解为先作乘法。

空行在隔开相邻的小程序段落方面的作用也是明显的。

移行也称向右缩格。它是指程序中各行不必都在左端对齐，都从第一格起排列。因为这样做使程序完全分不清层次关系。例如，两重选择结构嵌套，写成下面的移行形式，层次关系就清楚得多：

```
IF……THEN
   IF……THEN
   ……
```

```
    ELSE
    ……
    ENDIF
    ……
ELSE
    ……
ENDIF
```

④ 数据说明的次序

每个程序或模块在其可执行部分的前面都集中了一些说明语句。原则上，这些说明的次序与语法无关，其次序是任意的。但出于阅读理解和维护的要求，最好使其规范化，使说明的先后次序固定，如

常量说明

简单变量类型说明

数组说明

公用数据区说明

文件说明

在类型说明中还可进一步要求，比如，应按如下顺序排列：

整型量说明

实型量说明

复型量说明

逻辑量说明

同时，还要求在每一个说明语句的说明符之后，各个数据名按字典顺序排列。这样做，对于方便查找和修改都是十分有利的。例如，把

INTEGER SIZE, LENGTH, WIDTH, COST, PRICE

写成

INTEGER COST, LENGTH, PRICE, SIZE, WIDTH

⑤ 语句结构

为达到程序清晰，便于阅读和理解的目的，应做到

·每句至少占一行，不要多句一行。

·避免使用否定的逻辑条件。IF(NOT(A＞B))如在句中出现，应改写成 IF(A＜＝B)

·不要在语句中使用难于理解的技巧，防止给后面开发和维护工作带来麻烦。

·复杂的表达式最好利用括号表明运算的优先性，例如，A * B* *C 最好写成 A *(B* *C)。以免误解。

·不要片面地追求执行速度，而忽视了程序的简明和清晰性。事实上由于程序不清晰可能引起的错误，会延误得到正确结果，也就谈不到执行速度。

·充分利用库中现有的函数。

·尽量少用临时变量。

·如果使用的逻辑表达式不够直观，难于理解，应先将它简化。

·避免使用浮点数(实数)作相等比较。

此外，在 B. Kernighan 和 P. Plauger 写的《编程风格要点》一书中还提到：

·避免使用过于相似的变量名

·变量名中尽量不含数字

·同一变量名不要具有多种意义

·显式说明所有的变量

·表达式计算中注意浮点运算的误差

·注意整数运算的特点

·尽量少用语句标号

·优先考虑程序的正确性，然后再要求一定的运算速度

·不要为贪图效率而破坏了程序的清晰性

⑥ 输入和输出信息

输入和输出信息更是与用户的使用直接相关的。输入和输出的方式与格式应当尽可能方便用户的使用，一定要避免因设计不当给用户使用带来麻烦。例如，在一个程序或一个系统中，输入、输出格式应尽可能统一，在程序的注释行中给出输出数据的格式说明等。以交互式输入为例，应使输入步骤和操作尽可能简单，在输入数据的过程中及输入结束时都应在屏幕上给出状态信息。要根据软件用户的不同对象和特点(如对象是初学者或是专家还是一般用户)和不同的要求设计输入方案。输入数据的格式也力求简单，并应有完备的出错检查和出错恢复措施。采用菜单方式驱动对初学者或一般用户也许是合适的，但对熟练的专家，总让他用菜单可能会使他感到厌烦。命令方式如果设计得当可能被广泛接受。近年来软件开发中，对给用户提供怎样的使用手段给予了很大的重视，最终希望得到良好的用户界面。

(3) 程序实例

这里为读者提供一个程序的实例，希望这个实例能使读者对本章前面讨论的两个方面有进一步的理解和体会。

```
C * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C *                                                                       *
C *   SYSTEM NAME:     STUDENT MANAGEMENT SYSTEM                          *
C *                                                                       *
C *   MODULE NAME:     CKSS                                               *
C *                                                                       *
C *   PROGRAMMER:      LIU WUNG                                           *
C *                                                                       *
C *   DATE:            1985. 4. 20                                        *
C * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C
      SUBROUTINE CKSS (INPUT, INTMX, SSNUM, SSSIZ, *)
C
      INCLUDE 'CKSS. DOC'
C
C
C     SUBROUTINE DESCRIPTION
```

```
C
C
C     CKSS is the subroutine which extracts the student number
C     from the input string if the number is in the legal form
C     xxx—xx—xxxx, and all x's are integers.  An alternate
C     return will occur if it is not.
C
      INCLUDE 'CKSS. ARG'
C
C
C     ARGUMENT DESCRIPTION
C
C     INPUT—is the input string containing the SS#.
C          (input; CHARACTER INPUT * ( * ))
C     INTMX—is an integer variable containing the length of the
C          input string.
C          (input; INTEGER)
C     SSNUM—is the character string containing the extracted
C          SS#. from the input string.
C          (output; CHARACTER SSNUM * ( * ))
C        SSSIZ—is the size of the SSNUM string used to store
C          the SS#. This size is defined in parameter
C          P$SSMA.
C          (input; INTEGER)
C     * ———is the alternate error return if there are
C          illegal characters in the student number.
C          (output; LABEL #)
C
      INCLUDE' CKSS. LOC'
C
C
C     LOCAL VARIABLE DESCRIPTIONS
C
C
C     (NONE)
C
      IMPLICIT INTEGER(A—Z)
      INCLUDE' SYSIO. CM'
C
C     This common defines the system input and system output
C     logical unit number.  This is set in BLOCK DATA IGLOBK.
C
```

```
C       /SYSIO/
C       SYSIN—is an integer variable containing the logical unit
C            number for system input.
C       SYSOUT—is an integer variable containing the logical
C            unit number for system output.
C
        COMMON/SYSIO/SYSIN/SYSIN, SYSOUT
        INTEGER SYSIN, SYSOUT
        LOGICAL CKNUM
        CHARACTER INPUT * ( * ), SSNUM * ( * )
        SSNUM=''
        DSFLG=0
        DO 100 I I=1, INTMX
           L=II
C—
C— CHECK IF ZZZ OF ZZZ—XXX—XXX ARE NUMERIC
C—
        DO 500 I=1,3
           K=I
           IF (.NOT. CKNUM (INPUT(L:L))) GOTO 1000
           SSNUM(K:K)=INPUT(L:L)
           L=L+1
  500   CONTINUE
C—
C—CHECK IF THE'—'OR''OR A NUMERI C; XXX—XX—XXXX OR
C—XXX XX XXXX OR XXXXXXXXX
C—IF THE STARTED THE SS# IN THE FORM XXX—XXX THEN
C—IF MUST CONTINUE IN THE FORM XXX—XX—XXX
C—
        IF(INPUT(L:L). NE.'—'. AND. INPUT(L:L). NE.'') THEN
          IF (CKNUM (INPUT(L:L))) THEN
                K=K+1
                SSNUM(K:K)='—'
          ELSE
                GO TO 1000
          ENDIF
        ELSE
          K=K+1
          SSNUM(K:K)='—'
          L=L+1
          DSFLG=1
        ENDIF
C—
```

```
C—CHECK IF ZZ OF XXX—ZZ—XXXX ARE NUMERIC
C—
        DO 600 I=1,2
                K=K+1
          IF (. NOT. CKNUM(INPUT(L:L))) GO TO 1000
                SSNUM(K:K)=INPUT(L:L)
                L=L+1
  600         CONTINUE
C—
C—CHECK IF THE'—' OR ''OR A NUMERIC; XXX—XX—XXXX
C—OR XXX XX XXXX OR XXXXXXXXX
C—IF THE STARTED THE SS# IN THE FORM XXX—XX THEN
C—IF MUST CONTINUE IN THE FORM XXX—XX—XXXX
C—
        IF(INPUT(L:L). NE.'—'. AND. INPUT (L:L). NE. '') THEN
          IF (DSFLG. EQ. 1) GO TO 1000
          IF (CKNUM(INPUT(L:L))) THEN
               K=K+1
            SSNUM(K:K)='—'
          ELSE
            GO TO 1000
          ENDIF
        ELSE
          K=K+1
          SSNUM(K:K)='—'
          L=L+1
        ENDIF
C—
C—CHECK IF ZZZ OF XXX—XX—ZZZZ ARE NUMERIC
C—
        DO  700 I=1,4
            K=K+1
            IF(. NOT. CKNUM (INPUT(L:L))) GO TO 1000
            SSNUM(K:K)=INPUT (L:L)
            L=L+1
  700         CONTINUE
C—
C—IF DO NOT HAVE TRAILING BLANKS THEN ERROR
C—MESSAGE AND RETURN ELSE JUST RETURN
C—
        IF (L. GE. INTMX) RETURN
        IF (INPUT(L:INTMX). NE. '') GO TO 1000
            RETURN
```

```
  100     CONTINUE
C—
C—ILLEGAL CHARACTER IN SS#
C—
 1000     CONTINUE
          WRITE(SYSOUT, 2000)K
 2000     FORMAT ('*  *  * ILLEGAL CHARACTER IN
    +              SS NUMBER NEAR COLUMN',
    +              12, ' *  *  * ')
          RETURN
          END
```

第六章　软件测试

测试工作在软件生存期中占有重要位置。这不仅是因为测试阶段占用的时间、花费的人力以及成本的开销占软件生存期很大的比重，而且测试工作完成得怎样直接影响着软件的质量。如果在测试阶段未能把握住质量关，开发的软件质量得不到保证，在运行中很可能对整个系统造成十分严重的后果。

测试不完备给软件产品带来的损失有时是不可估量的。软件中隐含的缺陷如果未能在测试阶段及时发现，并加以解决，致使其混入运行阶段，轻则影响到系统的正常工作，重则导致整个系统的瘫痪，乃至造成无可挽回的事故。银行的存款额可能被化为乌有，甚至弄成赤字；学生学习成绩统计系统由于只接收两位十进制数字，因而把 100 分当成了 0 分。比较起来，这些事例还不算严重，让我们来看 1963 年发生在美国的实例。一个 FORTRAN 程序的循环语句

```
DO      5      I=1，3
```

被误写为

```
DO      5      I=1．3
```

由于空格对 FORTRAN 编译程序没有实际意义，误写的语句被当作了赋值语句

```
DO5I=1.3
```

这里“，”被误为“．”，一点之差致使飞往火星的火箭爆炸，造成一千万美元的损失，对于载人飞行器等控制系统的软件，由于是人命关天的项目，其质量就更显得重要了。

正是这个原因，在许多软件项目的测试阶段，投入了比开发

人员多出一倍的人力，并且要花去软件开发其它阶段所用资金的三倍到五倍，还要组织多个层次的检验，最终希望做到软件工作起来万无一失。

在你所参加的软件开发项目中，是否对软件测试给予足够的重视了，你是怎样作好测试工作的？本章准备为读者阐明一些软件测试的重要概念，介绍一些比较实用的测试方法，可供参考。

6.1 软件测试的基本概念

在讨论测试方法以前，需要澄清几个有用的并且比较重要的概念。

（1）软件测试的目的

软件开发项目在经历了计划、需求分析、设计和编码以后，已经取得了一些阶段成果，但是这些阶段成果能不能真正满足用户提出的需求，或者说它能在多大程度上满足用户需求，这是软件人员、管理人员以及用户都十分关心的问题。因为，大量的人力、物力投入了开发工作，又经历了阶段复审，人们切望拿到合格的成果。这时一系列的质量检验活动显然成为非常必要的了。

然而，软件开发项目的大量实践表明，这些“成果”常常是很不理想的。仅以编写出的源程序为例，可能遇到许多种情况。比如：

① 程序编写得无语法错误。这是程序编写是否正确的最基本要求。我们知道，具有语法错误的程序在上机运行时，无法通过编译系统语法检查这第一关。编译程序会列举出被运行程序的各种语法错误现象，而拒绝编译和执行。

② 程序执行中未发现明显的运行错误。这是指程序运行过程中没有因产生过大或过小的数据，由于溢出而无法继续执行；也没有遇到循环不已而障碍运行等情况。

③ 程序中没有不适当的语句。比如，有的变量未经说明而引

用，有的虽已作说明，却未曾引用，或者有的变量未赋初值而引用，以及有的变量被多次赋值，并未引用等等。

④ 程序运行时，能通过典型的有效测试数据，而得到正确的预期结果。即程序能接受规格说明所规定正常条件下的合理数据，并给出正确结果。

⑤ 程序运行时能通过典型的无效测试数据，而得到 正确的结果。即当程序接受规格说明所规定异常条件下的不合理数据时，能给出恰当的结果。

⑥ 程序运行时，能通过任何可能的数据，并给出正确的结果。

以上所列几种情况表明了程序正确性的差别。我们编写的程序如果能达到第六条要求，是很不容易的。第六条要求意味着程序中没有任何隐含的缺陷或差错。这是我们编写程序希望达到的最高目标。实际上，要达到这个目标必须付出相当大的代价。

十分明显，提高程序的正确性就是要尽可能发现和消除程序中隐藏的各种差错。

如上所述，编译系统接受用户的源程序后所作的语法检查能够发现程序编写时出现的语法错。但也仅仅是一些语法错。更多情况的差错编译系统无法查出。比如，程序中往往会出现的逻辑性差错、名字拼写错、不正确的初始化或未作初始化、数据格式或文件格式不对、循环次数有错、调用了错误的程序块或者纯属语义上的差错等等。

尽管程序正确性证明作为计算机科学的一个新分支，近年来取得了显著的发展，然而，尚未到达实用化阶段。要想解决以上列举的程序中的各种问题，目前比较实际的办法只能依靠测试技术。程序测试的目的是为了发现隐藏在程序内部的各种错误，有时也称为隐错(bug)。程序测试工作是指为发现程序错误而进行的各种活动。

也许有人认为，程序测试的目的是为了说明程序是没有问题的。在程序编写完以后，只需找到几个数据，使程序能够走通就达到目的了。我们认为，这种认识不仅不正确，而且常常是十分有害的。因为，若是出于这一目的，人们会自觉或不自觉地寻找容易使程序通过的测试数据，回避那些易于暴露程序错误的测试数据。致使隐藏的错误不被发现，自然也就得不到排除。与此相反，如果我们测试活动的目标始终围绕着揭露程序中的错误，那么在选取测试数据时，自然要考虑那些易于发现程序错误的数据。并且认为，能够发现程序错误的数据是好的数据，能够高效率揭露程序错误的测试是成功的测试。持相反观点的人必然认为那些是坏的数据，找出程序隐错的测试是失败的测试。

前去医院看病的病人总是希望早些作出诊断，弄清病因，以便对症施治。但如果病人讳疾忌医，在医院里虽作了一些检查，但未作出诊断，或医生只是随便作了一些检查，就告诉病人没病，病人自己可能一时心欢，却蕴蓄着危险的后患。就医院来说，这是对病人不负责任的做法。也许医生的医疗水平是值得怀疑的。不管怎样，这样的检查决不能认为是成功的检查。

这里当我们以前去医院就诊比喻时，有人可能提出疑问。也许会说，病人前去就诊是以有病为前提的，而写出的程序在测试以前能否肯定其中必有错误呢？实践表明，这个问题的回答是肯定的。其道理可从本章后面的讨论中弄清楚。

(2) 软件测试的对象

前面谈到，查找程序中的差错是测试工作的目的。但必须注意，软件测试并不等于程序测试。在软件测试阶段我们应该集中精力查找进入项目开发以来可能发生的各种错误，因此，需求分析、概要设计、详细设计以及程序编写等各开发阶段所得到的开发资料，包括需求规格说明、概要设计说明、详细设计说明以及源程序都应该是软件测试的对象。软件测试不应仅限在程序测试

的狭小范围内，而置其它开发阶段的工作于不顾。关于这一点我们准备在本章的最后一节专题讨论。另一方面，还应看到，由于开发工作各阶段是互相衔接的，前一阶段工作中发生的问题如未得到及时解决，很自然地要影响到下一阶段。从源程序的测试中找到的程序错误不一定都是程序编写过程造成的。如果简单地把程序中的错误全都归罪于程序员的程序编写，未免冤枉他们。据美国一家公司的统计表明，在查找出的软件错误中，属于需求分析和软件设计的错误约占 64%，属于程序编写的错误仅占 36%。这都说明，对程序编写而言，它的许多错误是“先天的”。

其实，这个事实也是容易理解的。到软件测试时为止，开发工作已经经历了多个环节，每个环节都有可能发生问题。目前我们还没有办法把握这些环节，使之不发生任何差错。图 6.1 表明了软件开发中一些可能造成差错的环节。在对需求理解和表达的正确性、设计和表达的正确性、实现的正确性以及运行的正确性中，任何一个环节上发生了问题都可能在软件测试中表现出来。

正是由于这个原因，我们通常使用“软件测试”，而不采用“程序测试”的提法。

(3) 软件测试的原则

以下提出的几条软件测试原则表面上看来似乎是显然的，然而却常常被人们所忽视。

① 在测试工作开始以前，不要设想程序中没有错误或查不出错误。除去上述软件开发多环节的特性以外，还应注意到原始问题的复杂性、软件本身的复杂性和抽象性，以及参加开发各种层次人员之间工作的配合关系等因素可能给软件带来的差错。固然，要求软件开发人员的思维应该很严密是合理的，并且也应该在开发的过程中采取一些措施，以防止发生差错。但人类思维的严密性毕竟是有限度的，开发过程中可能采取的可靠性措施也不是绝对有效的。它都无法完全杜绝软件差错的发生。对问题理解

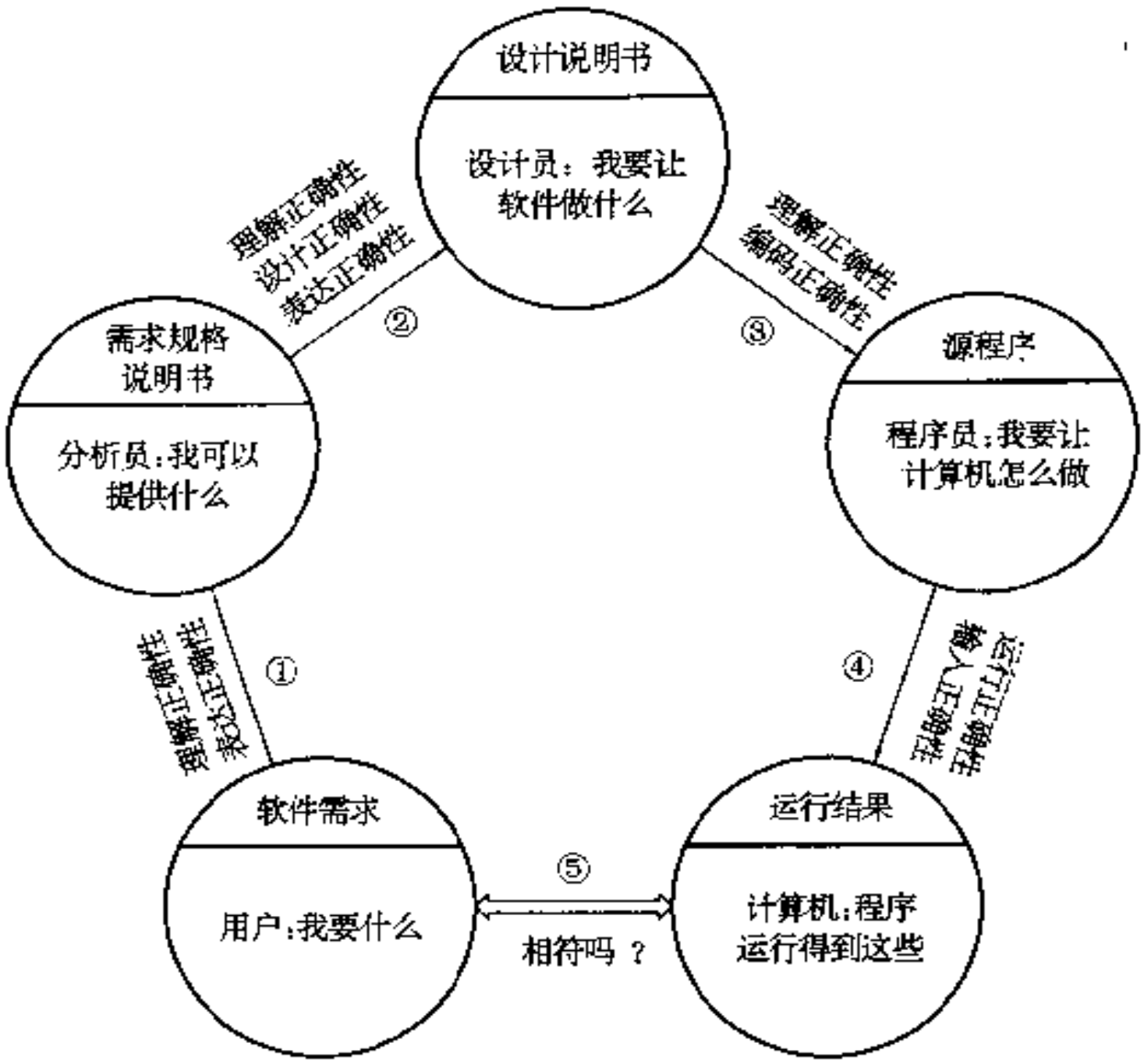


图 6.1　软件开发重要环节之间需要保持的正确性

上的差异、表达得不够确切、人员配合上不够协调、书写或按键的疏忽以至不可避免的修改会产生新的差错等等，经常会在软件开发的实践活动中出现，并且隐蔽得令人难以发现。对于这一点许多软件人员有着丰富的经验和深刻的教训。我们切不可抱有侥幸心理，以为写好的程序不经测试即能投入使用。

② 测试以前应预先确定测试的结果。要把测试结果当作测试用例的一个组成部分，要在选定测试数据的同时，确定预期的测试结果。在设计测试用例时，人们往往只注意选取测试数据，而忽略了这个测试数据应得到什么样的测试结果。殊不知有了预期结果，对于发现错误是会有帮助的。把准备好的精确写出的预

期结果与测试的输出结果相对照，便很容易发现问题。相反，如果事先并不知道什么是预期的结果，常常会把看起来似是而非的结果当成正确的结果，以致对测试的结果作出错误的判断，把本来应该发现的问题放过。

③ 程序人员尽可能避免测试自己编写的程序，程序开发小组也应尽可能避免测试本小组开发的程序。如果条件允许，最好建立独立的软件测试小组或测试机构。其原因在于，测试工作需要的是严格的精神、客观的态度和冷静的情绪。做好测试工作不能凭主观的臆想和愿望。实际上，人们常常由于各种原因具有一种不愿否定自己工作的心理，认为揭露自己程序中的错误总不是一件愉快的事。"闻过则喜"的要求难于让一般人做到。显然，这一心理状态会成为完成自己编写程序测试工作的障碍。正如工厂产品的检验工作一样，应有专职的检验人员负责。假定哪个工厂把产品质量的检验任务完全交给生产工人，其结果势必导致产品质量的大幅度下降。学术论文或著作的编写者与审阅者的关系也是类似的。

④ 测试用例的设计必须兼顾有效输入与无效输入。在测试程序时，人们常常只注意到合法的和想像得到的输入情况，而忽视了那些不合法的和预想不到的输入情况。实际上，一个软件在投入运行以后，一些意外的输入常常是会遇到的。最简单的例子是用户在键盘上按错了键或打入了非法命令。如果我们开发的软件遇到这种情况不能作出适当的反应，给出相应的信息或是给出了无意义的信息，都不能说这个软件可以正常工作了。显然，软件系统处理非法命令的能力也必须在测试时受到检验。在按意外的方式使用开发了的软件时，常常会发现一些问题，有时会比用合法的方式进行测试，其查错的收获还要大些。

⑤ 检验一个程序是否做了该做的事，仅仅是测试的一个方面，另一方面还要检验程序是否做了不该由它做的事。我们测试

程序时，必须注意检查是否有多余的副作用。通常情况下，多余的工作是无意义的，而且有时还是有害的。

⑥ 测试时不要被开始发现的若干错误所迷惑，找到了几个错误就以为问题已经解决了，不需要继续测试了。经验表明，程序中存在错误的概率和已经发现错误的个数大致成正比关系。图6.2的曲线表示了这个关系。

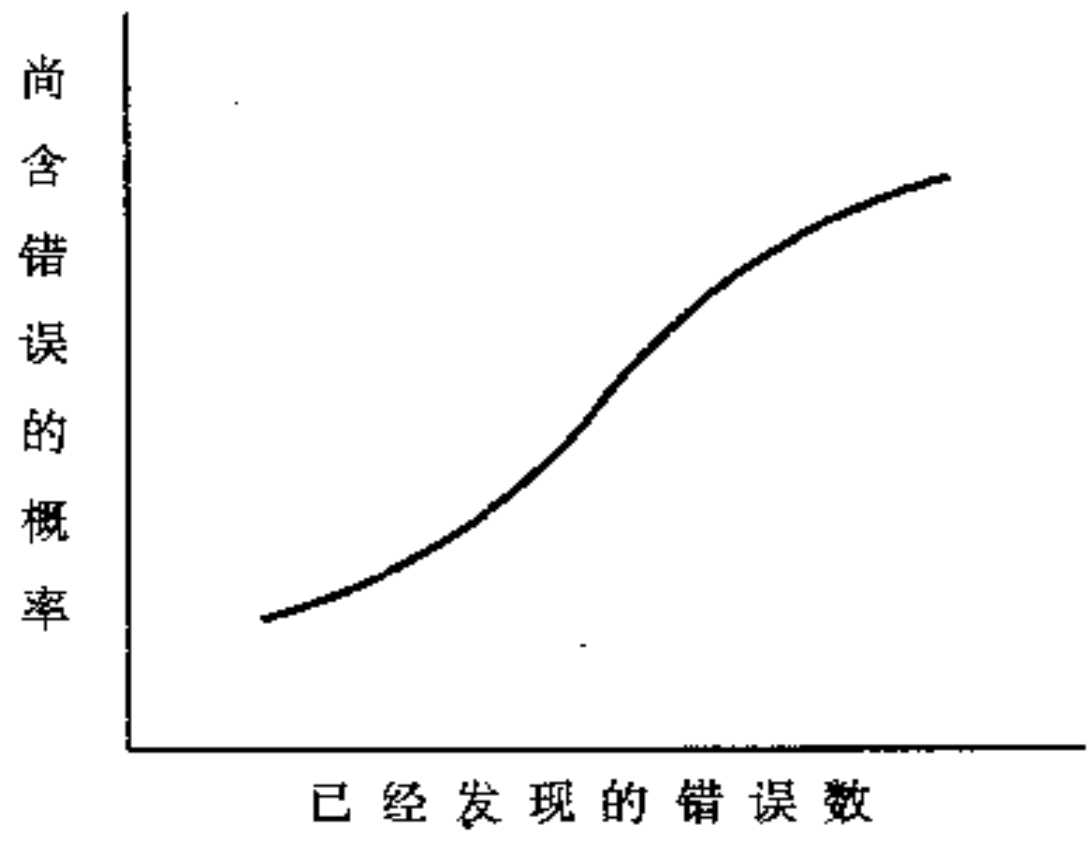


图 6.2　已发现与尚含错误概率的关系

这个事实还可用米中含砂的情况作比喻，当我们发现手中抓到的一把米里有砂时，绝不表示米袋中混入的砂粒仅此几个。并且如果有两袋米，各抓出一把，其中多砂的一把，预示着那一袋中的含砂量要更多些。这是显而易见的。

⑦ 测试工作限于人力和物力条件，只能进行到一定程度，适可而止。过度测试常常是不合算的。图 6.3 中的两曲线分别表明了测试的成本随着测试工作的进展而提高，同时，尚未找出的程序错误个数则逐渐减少。这两条曲线的交点指示了测试工作的截止点。如果超过此点，继续进行测试，那将要付出非常高的成本，花了高昂的代价，而其成效却是微小的。

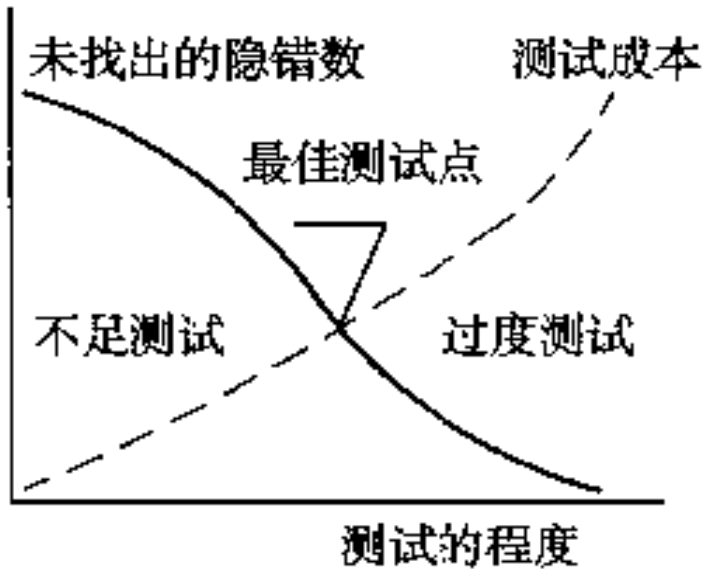


图 6.3　测试成本曲线

⑧ 测试完成后，应注意保存测试用例，而不要随意将其丢掉。这些测试用例无论是否发现了程序错误，都是经过精心设计的。保留已用过的测试用例，将会给重新测试带来方便，对于编写测试结果报告也是重要资料。

(4) 测试中遇到的错误类型

测试过程中发现的错误可能是各式各样的，这里仅按错误发生的影响和后果以及错误发生的性质和范围分别加以说明。

按错误发生的影响和后果，可能区分以下几种类型：

① 较小错误：这类错误只是对系统的输出结果有一些非实质性影响，比如，输出的数据格式不符合要求等。

② 中等错误：对系统的运行有局部的影响。如输出的某一部分数据有错误或出现冗余。

③ 较严重错误：系统的行为由于错误的干扰而出现明显不合情理的现象。如开出 0.00 元的支票，系统的输出结果完全不可信赖。

④ 严重错误：系统运行不可跟踪，一时不能掌握其规律，时好时坏。

⑤ 非常严重的错误：系统运行中突然停机，其原因不明，且无法软起动。

⑥ 最严重错误：运行被测的软件导致环境遭到破坏，或是造成事故，引起生命、财产的损失。

以上只是列举一些测试可能发生的现象，但我们可以通过这些现象来估计错误的严重程度，作到心中有数。

我们还可按错误的性质和范围，把出现错误的情况加以区分：

① 功能错误：由于功能规格说明书不够完整，或是叙述得不够确切，致使系统在实现时对功能有误解，例如给出了错误的功能、缺少了某些功能或是多出了冗余功能。

② 系统错误：例如发生以下各种情况：与外部接口打交道的协议错误、参数调用错误、子程序调用错、输入输出地址错、中断处理错等等。此外还可能有与操作系统接口错误、控制顺序错误以及资源管理的问题等。

③ 过程错误：如算术运算错误、初始化错误及逻辑错误等。

④ 数据错误：例如数据内容、结构与属性错误，动态数据与静态数据混淆或是参数与控制数据混淆等。

⑤ 编码错误：如语法错、变量名错、局部量与全局量混 淆或程序逻辑错误等。

对于普通的软件系统，有人按上述错误的类型作了统计，表6.1 表明不同类型错误所占的比例。

表 6.1　软件系统的错误类型

错误类型	功能错	系统错	过程错	数据错	编码错	其它
百分比	27%	16%	27%	10%	4%	16%

(5) 测试信息流

图 6.4 给出的测试信息流能够大致反映出整个测试工作的主

要活动。图中表明，软件测试需要三种信息：

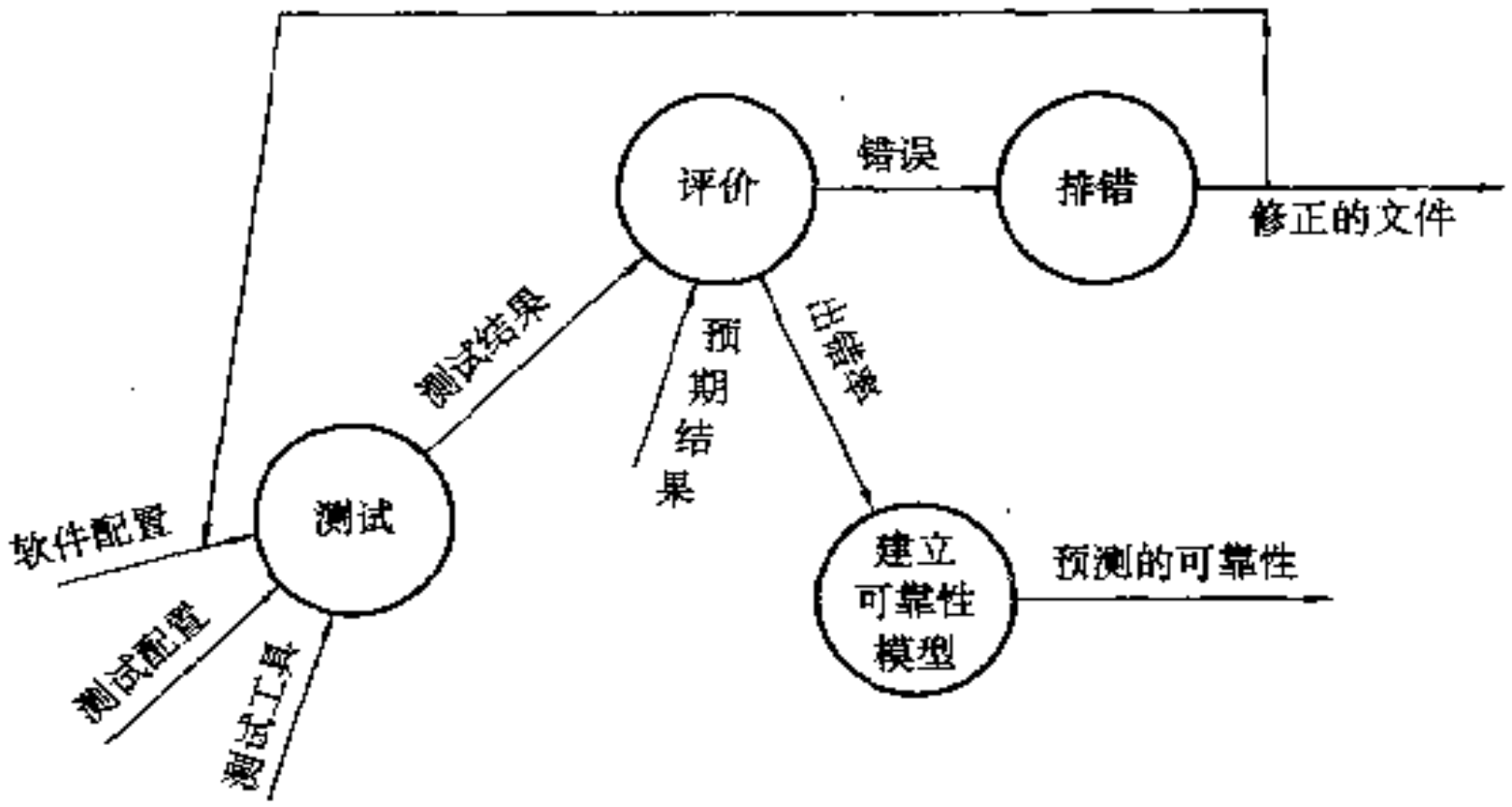


图 6.4　测试信息流

① 软件配置：这是被测的文件，它包括被开发软件的需求规格说明书、设计说明书以及源程序等。

② 测试配置：这是测试必不可少的信息，它包括，表明测试工作如何进行的测试计划；给出测试数据的测试用例以及控制测试进程的测试程序。

③ 测试工具：为提高软件测试的效率，测试工作需要有软件测试工具的支持，这些测试工具实际上是一些支撑软件，它们的工作都是从某些方面服务于测试的进行，以减轻人们完成测试任务中的手工劳动。例如，用本章第 2 节和第 3 节讨论的方法均可开发出相应的工具支持测试工作。

把测试结果与预期结果进行比较后可进行测试评价，根据出错的迹象，找到错误的准确部位，进行排错，同时修正相关的文件。修正后的文件一般都需要经过再次测试，直至测试通过为止。另一方面，根据出错的信息建立被开发软件的可靠性

模型，得到的可靠性数据对于软件投入运行后的维护工作是有意义的。

(6) 测试与调试

人们常常把测试(testing)与调试(debugging)混为一谈；实际上它们具有完全不同的含意。测试是决定程序中确有某种错误的过程，完全不包含改错的工作。调试是在完成测试以后，准确判定错误位置以及具体的出错情况，继而进行修正，以排除错误。调试有时也被称为排错或纠错。

进行测试时，我们是借助于测试结果与预期结果之间的差异来确定错误存在的。但准确判定错误的位置并不是一件容易的事，它要占去调试工作的大部分工作量(约90%)。我们可以把测试比作对人体进行各项检查和化验，把检查和化验结果进行分析便可肯定有病在身。调试则相当于作出诊断，确定在什么部位有什么病，并给予治疗，直至痊愈。实际上，诊断往往是很困难的工作。

(7) 穷举测试和测试的缺欠

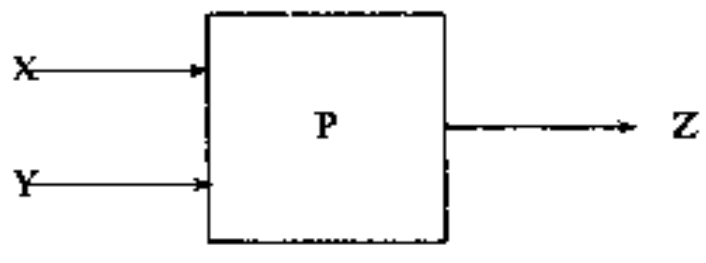


图 6.5

既然测试的目的在于寻找错误，并且找出的错误越多越好。很自然就会提出这样的问题，能不能把所有隐藏的错误全部找出来呢？或者说能不能把所有可能做的测试无遗漏地一一做完来找出所有的错误呢？下面按两种常用的测试方法作出具体的分析：

① 黑盒测试

黑盒测试也称功能测试或数据驱动测试。用这一方法进行测试时，程序被看作不能打开的黑盒。在完全不考虑程序内部结构和内部特性的情况下，测试者只能依靠程序需求规格说明书，从

可能的输入条件和输出条件中确定测试数据。也就是根据程序的功能或程序的外部特性设计测试用例。

若一程序 P 有输入量 X 和 Y 及输出量 Z(图 6.5)，在字长为 32 位的计算机上运行。如果 X 和 Y 均只取整数，考虑把所有可能的 X，Y 值作为测试数据，按黑盒方法进行穷举测试，力图全面、无遗漏地“挖掘”出程序中的所有错误。

这样做可能采用的测试数据组 X_i 和 Y_i，基中 i 的最大值为

$$2^{32} \times 2^{32} = 2^{64}$$

如果程序 P 测试一组 X、Y 数据需要 1/1000 秒，要完成 2^{64}组测试，需用 5 亿年的时间。

② 白盒测试

白盒测试又称结构测试或逻辑驱动测试。这种测试允许测试者考察程序的内部结构，并根据程序的内部结构设计测试数据，而完全不顾程序的功能。

若一程序有四个条件判断(图 6.6 中 d_1、d_2、d_3 和 d_4)和一个最多重复 20 次的循环。在循环体内有五条路径。若考虑到循环，从入口到出口总的路径数是

$$5^{20} \approx 10^{14}$$

假定我们为每条路径设计一组测试数据，再对其实施测试，这一过程若是需要两分钟，则测试完 10^{14}条路径也要用 5 亿年的时间。

以上两种情况的分析表明，实行穷举测试由于工作量过大，需用的时间过长，实施起来是不现实的，也就失去了实用价值。我们知道，任何软件开发项目都要受到期限、费用、人力和机时等条件的限制，尽管我们以为为了充分揭露程序中的所有隐藏错误，彻底的作法是针对所有可能的数据进行测试，但事实告诉我们，这样做是不可能的。

软件工程的总目标是充分利用有限的人力、物力资源，高效

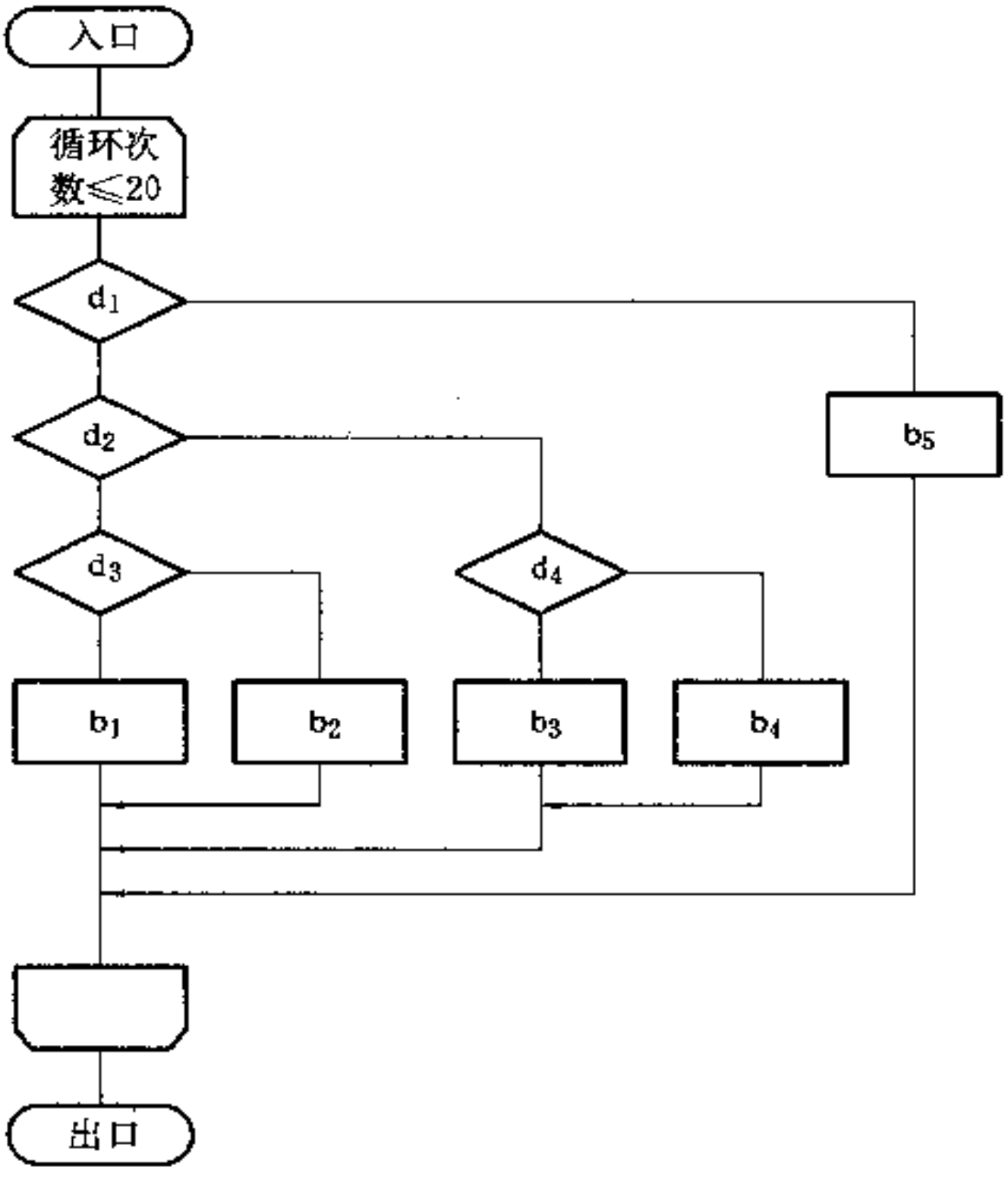


图 6.6

率、高质量、低成本地完成软件开发项目。在测试阶段既然穷举测试是不可实现的，为了节省时间和资源，提高测试效率，就必须精心设计测试用例，也就是要从数量极大的可用测试用例中精心地挑选少量的测试数据，使得采用这些测试数据能够达到最佳的测试效果，或者说它们能够高效率地把隐藏的错误揭露出来。

以上事实说明了软件测试具有一个致命的缺欠：由于任何程

序只能进行少量(相对于穷举的巨大数量而言)有限的测试，在发现错误时说明程序是有问题的，但在未发现问题时，并不能说明程序中不存在错误，不能说明程序没有问题。

6.2 测试用例的设计

既然测试工作不能采用穷举的办法，那么少量有效的测试用例如何选取变成了测试的关键性问题。必须排除选取测试用例的盲目性，最典型的盲目性表现是随机地任意选取的作法。即在程序的所有可能输入值中任意选取某些做为测试用例。显然，这样的做法没有任何针对性，难于达到高效和最优。

本节将介绍几种实用的测试用例设计方法。其中，逻辑覆盖属于白盒测试方法，等价类划分、边值分析及因-果图属于黑盒测试方法。

(1) 逻辑覆盖

采用逻辑覆盖原则设计测试用例进行测试也称为逻辑驱动测试，是从程序内部的逻辑结构出发选取测试用例的方法。使用这一方法要求测试人员对程序的逻辑结构有清楚的了解，甚至要能掌握源程序的所有细节。由于覆盖的目标不同，逻辑覆盖又可分为：语句覆盖、判定覆盖、条件覆盖、判定与条件覆盖及路径覆盖。以下将分别作出扼要的介绍。在所介绍的五种逻辑覆盖中，均以图 6.7 给出的流程图所表达的程序段为例。其中有两个判断，每个判断都包含复合条件的逻辑表达式，并且符号“AND”表示“与”运算，符号“OR”表示“或”运算。图中的 a、b、c、d 和 e 表示若干个程序点。

① 语句覆盖

语句覆盖的含意是，设计若干个测试用例，运行被测程序，使程序中的每个可执行语句至少执行一次。

在上述程序段中，我们如果选用的测试用例是

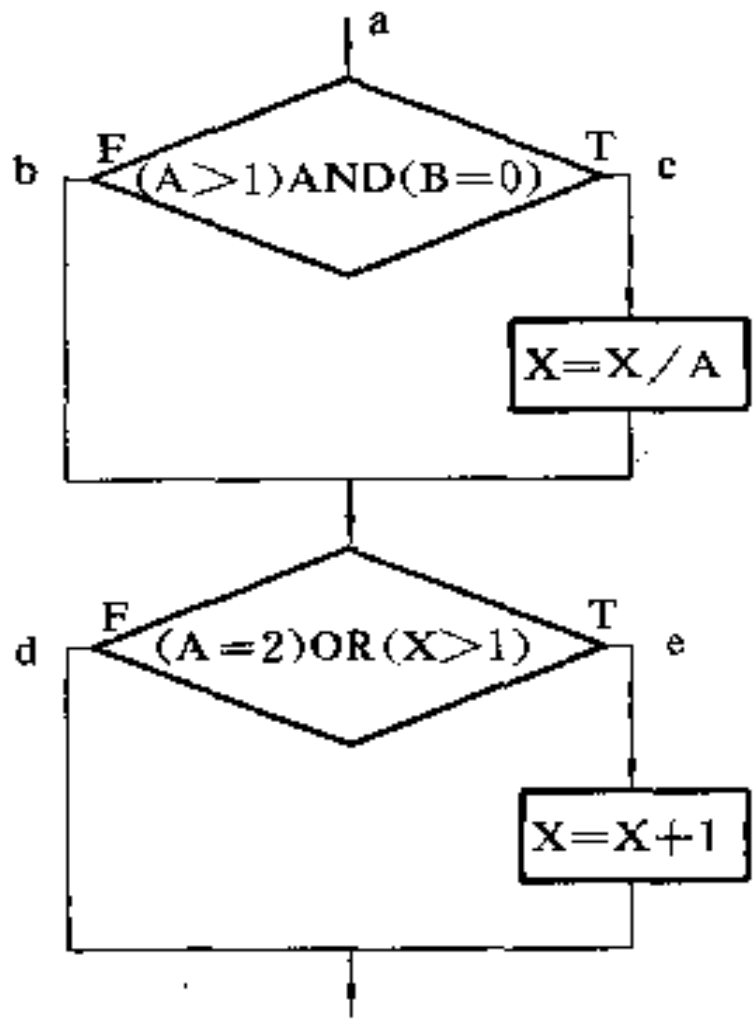


图 6.7　逻辑覆盖测试例

$$\begin{cases} A = 2 \\ B = 0 \text{ ---------- CASE1} \\ X = 3 \end{cases}$$

则程序按路径 ace 执行。这样该程序段 的四个语句均得到执行，从而作到了语句覆盖。但如要选用的测试用例是

$$\begin{cases} A = 2 \\ B = 1 \text{ ---------- CASE2} \\ X = 3 \end{cases}$$

程序按路径 abe 执行，便未能达到语句覆盖。

从程序中每个语句都得到执行这一点来看，语句覆盖的方法似乎能够比较全面地检验每一个语句。但它也绝不是完美无缺的。假如这一程序段中两个判断的逻辑运算有问题，例如，第一个判断的运算符“AND”错成了运算符“OR”或是第二个判断中的

运算符“OR”错成了运算符“AND”。这时仍使用上述前一个测试用例 CASE1，程序将仍按路径 ace 执行。这说明虽然也作到了语句覆盖，却发现不了判断中逻辑运算的错误。和后面介绍的其它覆盖相比较，实际上，语句覆盖是最弱的逻辑覆盖准则。

② 判定覆盖

按判定覆盖准则进行测试是指，设计若干测试用例，运行被测程序，使得程序中每个判断的取真分支和取假分支至少经历一次，即判断的真假值均曾被满足。判定覆盖又称为分支覆盖。

仍以上述程序段为例，若选用的两组测试用例是

$$\begin{cases} A = 2 \\ B = 0 \\ X = 3 \end{cases} \text{---------- CASE1}$$

$$\begin{cases} A = 1 \\ B = 0 \\ X = 1 \end{cases} \text{---------- CASE3}$$

则可分别执行路径 ace 和 abd，从而使两个判断的四个分支 c、e 和 b、d 分别得到覆盖。

当然，我们也可选用另外两组测试用例：

$$\begin{cases} A = 3 \\ B = 0 \\ X = 3 \end{cases} \text{---------- CASE4}$$

$$\begin{cases} A = 2 \\ B = 1 \\ X = 1 \end{cases} \text{---------- CASE5}$$

分别经历路径 acd 及 abe，同样也可覆盖四个分支。

我们注意到，上述两组测试用例不仅满足了判定覆盖，同时

还做到了语句覆盖。从这一点看似乎判定覆盖比语句覆盖更强一些，但让我们设想，在此程序段中的第二个判断条件 X＞1 如果错写成 X＜1，使用上述测试用例 CASE5，照样能按原路径执行(abe)，而不影响结果。这个事实说明，只作到判定覆盖仍无法确定判断内部条件的错误。因此，需要有更强的逻辑覆盖准则去检验判断内部条件。

以上仅考虑了两出口的判断，我们还应把判定覆盖准则扩充到多出口判断(如 CASE 语句)的情况。

③ 条件覆盖

条件覆盖是指，设计若干测试用例，执行被测程序以后，要使每个判断中每个条件的可能取值至少满足一次。

在上述程序段中，第一个判断应考虑到

A＞1　取真值，记为 T_1

A＞1　取假值，即 A≤1，$\overline{T}_1$

B＝0　取真值，记为 T_2

B＝0　取假值，即 B≠0，记为 $\overline{T}_2$。

第二个判断应考虑到

A＝2　取真值，记为 T_3

A＝2　取假值，即 A≠2，记为 $\overline{T}_3$

X＞1　取真值，记为 T_4

X＞1　取假值，即 X≤1，记为 $\overline{T}_4$

我们给出三个测试用例：CASE6，CASE7，CASE8，执行该程序段所走路径及覆盖条件是：

测试用例	A	B	X	所走路径	覆　盖　条　件
CASE 6	2	0	3	ace	T_1，T_2，T_3，T_4
CASE 7	1	0	1	abd	$\overline{T_1}$，T_2，$\overline{T_3}$，$\overline{T_4}$
CASE 8	2	1	1	abe	T_1，$\overline{T_2}$，T_3，$\overline{T_4}$

从这个表中可以看到，三个测试用例把四个条件的八种情况均作了覆盖。

进一步分析上表，覆盖了四个条件的八种情况的同时，把两个判断的四个分支 b、c、d 和 e 似乎也被覆盖。我样我们是否可以说，做到了条件覆盖，也就必然实现了判定覆盖呢？让我们来分析另一情况。假定选用两组测试用例是 CASE9 和 CASE8，执行程序段的覆盖情况是：

测试用例	A	B	X	所走路径	覆盖分支	覆盖条件
CASE 9	1	0	3	abe	be	$\overline{T_1}$, T_2, $\overline{T_3}$, T_4,
CASE 8	2	1	1	abe	be	T_1, $\overline{T_2}$, T_3, $\overline{T_4}$,

这一覆盖情况表明，覆盖了条件的测试用例不一定覆盖了分支。事实上，它只覆盖了四个分支中的两个。为解决这一矛盾，需要对条件和分支兼顾。

④ 判定-条件覆盖

判定-条件覆盖要求设计足够的测试用例，使得判断中每个条件的所有可能至少出现一次，并且每个判断本身的判定结果也至少出现一次。

例中两个判断各包含两个条件，这四个条件在两个判断中可能有八种组合，它们是：

1) $A>1$, $B=0$　　记为 T_1, T_2

2) $A>1$, $B\neq 0$　　T_1, $\overline{T_2}$

3) $A\leqslant 1$, $B=0$　　$\overline{T_1}$, T_2

4) $A\leqslant 1$, $B\neq 0$　　$\overline{T_1}$, $\overline{T_2}$

5) $A=2$, $X>1$　　T_3, T_4

6) A=2, X≤1　　T_3, $\overline{T_4}$

7) A≠2, X>1　　$\overline{T_3}$, T_4

8) A≠2, X≤1　　$\overline{T_3}$, $\overline{T_4}$

这里设计了四个测试用例，用以覆盖上述八种条件组合：

测试用例	A	B	X	覆盖组合号	所走路径	覆盖条件
CASE 1	2	0	3	1),5)	ace	T_1, T_2, T_3, T_4,
CASE 8	2	1	1	2),6)	abe	T_1, $\overline{T_2}$, T_3, $\overline{T_4}$,
CASE 9	1	0	3	3),7)	abe	$\overline{T_1}$, T_2, $\overline{T_3}$, T_4,
CASE 10	1	1	1	4),8)	abd	$\overline{T_1}$, $\overline{T_2}$, $\overline{T_3}$, $\overline{T_4}$,

我们注意到，这一程序段共有四条路径。以上四个测试用例固然覆盖了条件组合，同时也覆盖了四个分支，但仅覆盖了三条路径，却漏掉了路径 acd。前面讨论的多种覆盖准则，有的虽提到了所走路径问题，但尚未涉及到路径的覆盖，而路径能否全面覆盖在软件测试中是个重要问题，因为程序要取得正确的结果，就必须消除遇到的各种障碍，沿着特定的路径顺利执行。如果程序中的每一条路径都得到考验，才能说程序受到了全面检验。

⑤ 路径覆盖

按路径覆盖要求进行测试是指，设计足够多测试用例，要求覆盖程序中所有可能的路径。

针对例中的四条可能路径(图 6.8)

ace　　记为 L_1

abd　　记为 L_2

abe　　记为 L_3

acd　　记为 L_4

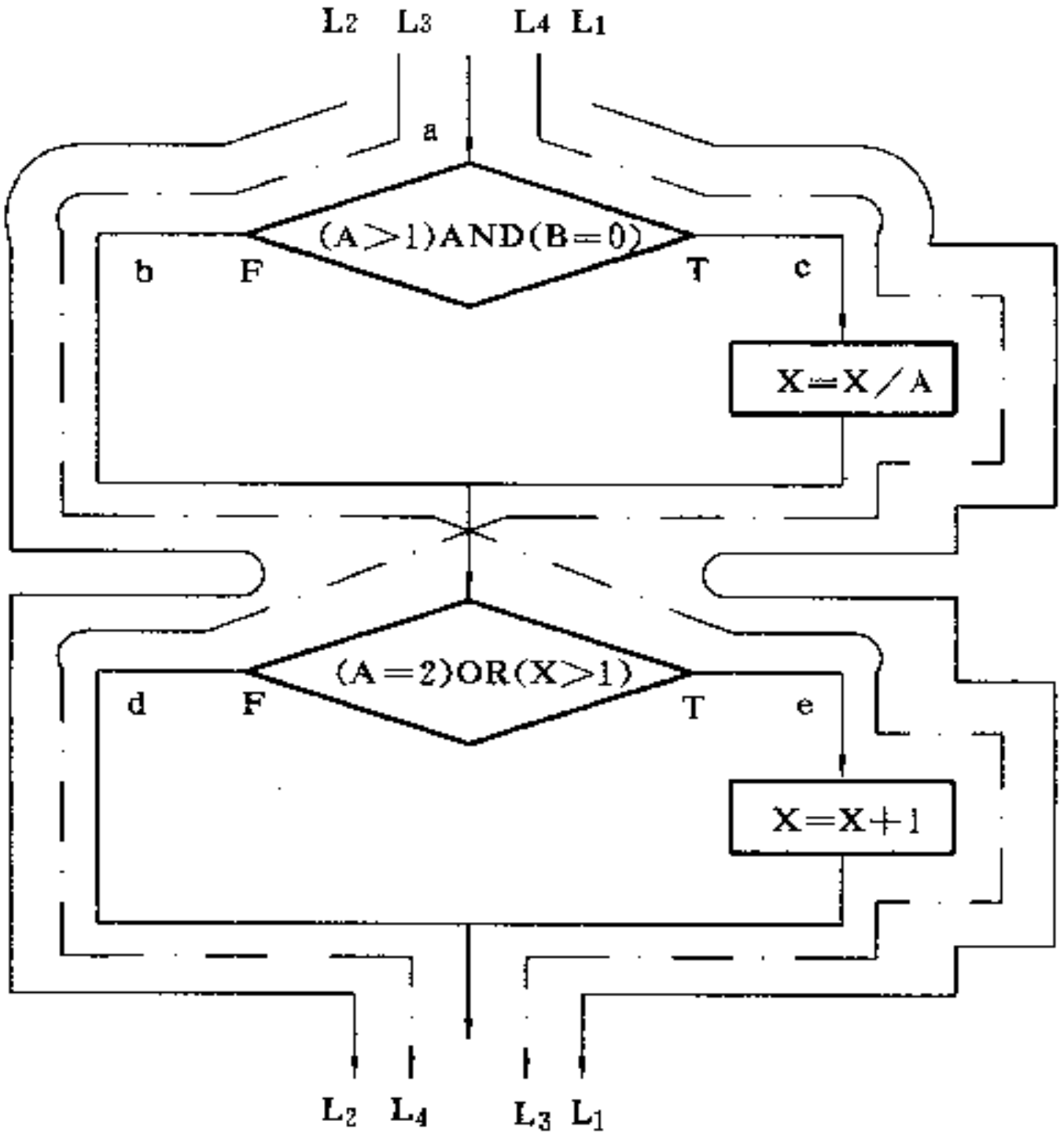


图 6.8

现给出四个测试用例：CASE1，CASE7，CASE8 和 CASE11，使其分别覆盖这四路径：

测试用例	A	B	X	覆盖路径
CASE 1	2	0	3	ace (L_1)
CASE 7	1	0	1	abd (L_2)
CASE 8	2	1	1	abe (L_3)
CASE 11	3	0	1	acd (L_4)

这里所用的程序段非常简短，也只有四条路径。但在实际问题中，一个不太复杂的程序，其路径数都是一个庞大的数字。在前面图 6.6 所示的程序竟有 5^{20} 条路径，要在测试中覆盖这样多的路径是无法实现的。为解决这一难题只得把覆盖的路径数压缩到一定限度内，例如，程序中的循环体只执行一次。

(2) 等价类划分

等价类划分是黑盒测试方法，它完全不考虑程序的内部结构，只根据程序的规格说明设计测试用例。

既然穷举测试数据的数量太大，实际上完成穷举测试的办法不可行，我们只能选取其中的一部分作为测试用例。问题在于如何选取。等价类划分的办法是把程序的输入域划分成若干部分，然后从每个部分中选取少数代表性数据当作测试用例。使用这一方法设计测试用例要经历划分等价类(列出等价类表)及选取测试用例两步。以下分别加以说明，然后给出实例。

① 划分等价类

首先把数目极多的输入情况划分成若干个等价类。所谓等价类是指某个输入域的集合。它表示，如果用集合中的一个输入条件作为测试数据进行测试不能发现程序中的错误，那么使用集合中的其它输入条件进行测试也不可能发现错误。也就是说，对揭露程序中的错误来说，集合中的每个输入条件是等效的。如果我们的测试数据全都从同一个等价类中选取，除去其中的一个测试数据对发现程序错误有意义以外，针对其余的测试数据进行的测试都是徒劳的，因为它们对测试工作的进展没有任何益处。我们不如把测试的时间花在其它等价类输入条件的测试中。例如，如果以 2 和 3 分别作为输入某一程序的两个测试数据，它们具有等价的测试效果(即如果 2 作为测试数据，在程序测试中能暴露某类错误，3 若作为测试数据也能发现同一类错误)。我们宁愿只取其中的一个作为测试数据，作一次测试，而不取两个，分别作两

次测试。

在考虑等价类时，应该注意区别两种不同的情况：

·有效等价类：有效等价类指的是对程序的规格说明是有意义的、合理的输入数据所构成的集合。在具体问题中，有效等价类可以是一个，也可以是多个。

·无效等价类：无效等价类指对程序的规格说明是不合理或无意义的输入数据所构成的集合。对于具体的问题，无效等价类至少应有一个，也可能有多个。

如何确定等价类，这是使用等价类划分方法的一个重要问题。以下结合具体实例给出几条确定等价类的原则：

1）如果输入条件规定了取值范围或值的个数，则可确定一个有效等价类和两个无效等价类。例如，程序的规格说明中提到的输入条件包括"…项数可以从 1 到 999…"，则可取有效等价类为"1＜项数＜999"。无效等价类为"项数＜1"及"项数＞999"。又如，程序规格说明中提到"…学生允许选修 2 至 4 门课…"，有效等价类可取"选课 2 至 4 门"，无效等价类为"只选一门课或未选课"及"选课超过 4 门"。

2）输入条件规定了输入值的集合，或是规定了"必须如何"的条件，则可确定一个有效等价类和一个无效等价类。例如，某程序的规格说明中提到的输入条件包括"…统计全国各省、市、自治区的人口…"，则应取"国内省、市、自治区"为有效等价类，"非国内省、市、自治区"为无效等价类。又如，某程序涉及到标识符，其输入条件规定"标识符应以字母开头…"，则"以字母开头"者作为有效等价类，"以非字母开头"为无效等价类。

3）如果我们确知，已划分的等价类中各元素在程序中的处理方式是不同的，则应将此等价类进一步划分成更小的等价类。

等价类划分完以后，可按下面的形式列出等价类表：

输入条件	有效等价类	无效等价类
…	…	…
…	…	…

② 确定测试用例

根据已列出的等价类表，按以下步骤确定测试用例

1）为每个等价类规定一个唯一的编号。2）设计一个测试用例，使其尽可能多地覆盖尚未覆盖的有效等价类。重复这一步，最后使得所有有效等价类均被测试用例所覆盖。

3）设计一个新的测试用例，使其只覆盖一个无效等价类。重复这一步使所有无效等价类均被覆盖。

注意，这里规定每次只覆盖一个无效等价类，因为一个测试用例中如果含有多个错误，有可能在测试中只发现其中的一个，另一些被忽视。例如，程序的规格说明中规定了“…每类科技用书 10 至 50 册，…”，若一个测试用例为“小说 5 册”，在测试中很可能只检测出书的类型错误，而忽略了书的册数错误。

③ 用等价类划分法设计测试用例实例

某 FORTRAN 编译系统的设计与程序编写已经完成，现需对其中实现 DIMENSION 语句的测试设计测试用例。已知 DIMENSION 语句的语法规则是：

“DIMENSION 语句用以规定数组的维数，其形式为：

DIMENSION AD[, AD]…

其中，AD 是数组描述符，其形式为

n(d[, d]…)

其中，n 是数组名，由 1—6 个字母或数字组成，为首的必须是字母；

d 是维数说明符。数组维数最大为 7，最小为 1。它的形式为

[lb：]ub

lb 和 ub 分别表示数组下标的下界和上界，均为－65，534 至 65，535 之间的整数，也可是整型变量名(但不可是数组元素名)。若未规定 lb，则认为其值为 1，且 ub≥lb。若已规定 lb，则它可为负数、零或正数。

DIMENSION 语句也和其它语句一样，可连续写多行。

以上规则中，小写字母代表语法单位。方括号内是任选项；省略号表示它前面的项可重复出现多次。”

按下面两步用等价类划分方法，设计测试用例。

第一步　确定输入条件，并确定等价类。见下页表。

第二步　确定测试用例。先设计一个测试用例，使它覆盖多个有效等价类。如

```
DIMENSION A(2)
```

能覆盖有效等价类 1，4，7，10，12，15，24，28，29 和 40。为覆盖其它有效等价类，需设计另外的测试用例。如

```
DIMENSION A12345(I, 9, J4X X X X, 65535, 1,
          KLM, 100), BBB (－65534：100, 0：1000,
          10：10, I：65535)
```

它能覆盖其余的有效等价类。

以下一个个地设计测试用例，使每个测试用例只覆盖一个无效等价类，直至覆盖完为止。这些测试用例是(下列各测试用例所覆盖的等价类号在其左端的括弧内给出)：

```
(3) DIMENSION
(5) DIMENSION   (10)
(6) DIMENSION   A234567(2)
(9) DIMENSION   A. 1(2)
(11) DIMENSION   1A(10)
(13) DIMENSION   B
```

输入条件	有效等价类	无效等价类
数组描述符个数	1(1),>1(2)	无数组描述符(3)
数组名长度	1～6(4)	0(5),>6(6)
数组名	有字母(7),有数字(8)	有其它字符(9)
数组名以字母开头	是(10)	不是(11)
数组维数	1～7(12)	0(13),>7(14)
上界是	常数(15),整型变量(16)	数组元素名(17),其它(18)
整型变量名	有字母(19),有数字(20)	其它(21)
整型变量名以字母开头	是(22)	不是(23)
上下界取值	-65,531～65,535(24)	<-65,534(25) >65,535(26)
是否定义下界	是(27),否(28)	
上界对下界关系	>(29),=(30)	<(31)
下界定义为	负数(32),0(33),正数(34)	
下界是	常数(35),整型变量(36)	数组元素名(37),其它(38)
语句多于一行	是(39),不是(40)	

(注：表中括弧内数字为等价类号码)

(14) DIMENSION B(4, 4, 4, 4, 4, 4, 4, 4)

(17) DIMENSION B(4, A(2))

(18) DIMENSION B(4, , 7)

(21) DIMENSION C(I. , 10)

(23) DIMENSION C(10, 1J)

(25) DIMENSION D(—65535 : 1)

(26) DIMENSION D(65536)

(31) DIMENSION D(4 : 3)

(37) DIMENSION D(A(2) : 4)

(38) DIMENSION D(. : 4)

以上共计 18 个测试用例，它们覆盖了全部等价类。

从这例中我们看到，使用等价类划分方法可以系统地、全面地考虑黑盒测试的测试用例的设计问题。

(3) 边值分析

在软件的设计和程序编写中，常常对规格说明中的输入域边界或输出域的边界不够注意，以致形成一些差错。实践表明，在设计测试用例时，对边界处的处理应给予足够的重视，为检验边界附近的处理专门设计测试用例，常常取得良好的测试效果。

比如，在作三角形计算时，要输入三角形的三个边长：A、B 和 C。我们应注意到这三个数值应当 满 足 A+B>C、 A+C>B 及 B+C>A 才能构成三角形。但如果把三个不等式的任何一个大于号“>”错写成大于等于号“≥”，那就不能构成三角形。问题恰恰出现在容易被疏忽的边界附近。这里所说的边界是指，相当于输入等价类和输出等价类而言，稍高于其边界值及稍低于其边界值的一些特定情况。

针对边界值设计测试用例时，应注意遵循以下几条原则：

① 如果输入条件规定了取值范围，或是规定了值的个数，则应以该范围的边界内及刚刚超出范围的边界外的值，或是分别对最大、最小个数及稍小于最小、稍大于最大个数作为测试用例。例如，如果程序的规格说明中规定：“重量在 10 千克至 50 千克范围

内的邮件，其邮费计算公式为…”。作为测试用例，我们应取 10 及 50，还应取 10.01，49.99，9.99 及 50.01 等。如果另一问题规格说明规定：“某输入文件可包含 1 至 255 个记录，…”，则测试用例可取 1 和 255，还应取 0 及 256 等。

② 针对规格说明的每个输出条件使用前面的第①条原则。例如，某程序的规格说明要求计算出“每日保险金扣除额为 0 至 1165.25 元”，其测试用例可 0.00 及 1165.25，还应取－0.01 及 1165.26 等。如果另一程序属于情报检索系统，要求每次“最多显示四条情报摘要”，这时我们应考虑的测试用例包括 1 和 4，还应包括 0 和 5 等。

③ 如果程序规格说明中提到的输入或输出域是个有序集（如顺序文件、表格等），就应注意选取有序集的第一个和最后一个元素作为测试用例。

④ 分析规格说明，找出其它的可能边界条件。

以下给出实例，说明在具体问题中，边界值是怎样考虑的。

一个为学生考试试卷评分和成绩统计的程序，其规格说明指出了对程序的要求：

“程序的输入文件由 80 个字符的一些记录组成，这些记录分为三组：

① 标题。这一组只有一个记录，其内容为输出报告的名字。

② 试卷各题标准答案记录。每个记录均在第 80 个字符处标以数字“2”。该组的第一个记录的第 1 至第 3 个字符为题目编号（取值为 1－999）。第 10 至第 59 个字符给出第 1 至第 50 题的答案（每个合法字符表示一个答案）。该组的第 2，第 3，……一个记录相应为第 51 至第 100，第 101 至 150，……题的答案。

③ 第三组记录描述了每个学生的答卷。该组中每个记录的第 80 个字符均为数字“3”。每个学生的答卷在若干个记录中给出。如甲的首记录第 1 至第 9 字符给出学生姓名及学号，第 10 至

第 59 字符列出的是甲所做的第 1 至第 50 题的解答。若试题数超过 50，则其第 2，3，……记录分别给出他的第 51 至第 100，第 101 至第 150……题的解答。然后是学生乙的答卷记录。

若学生最多为 200 人。输入数据的形式如图 6.9 所示。

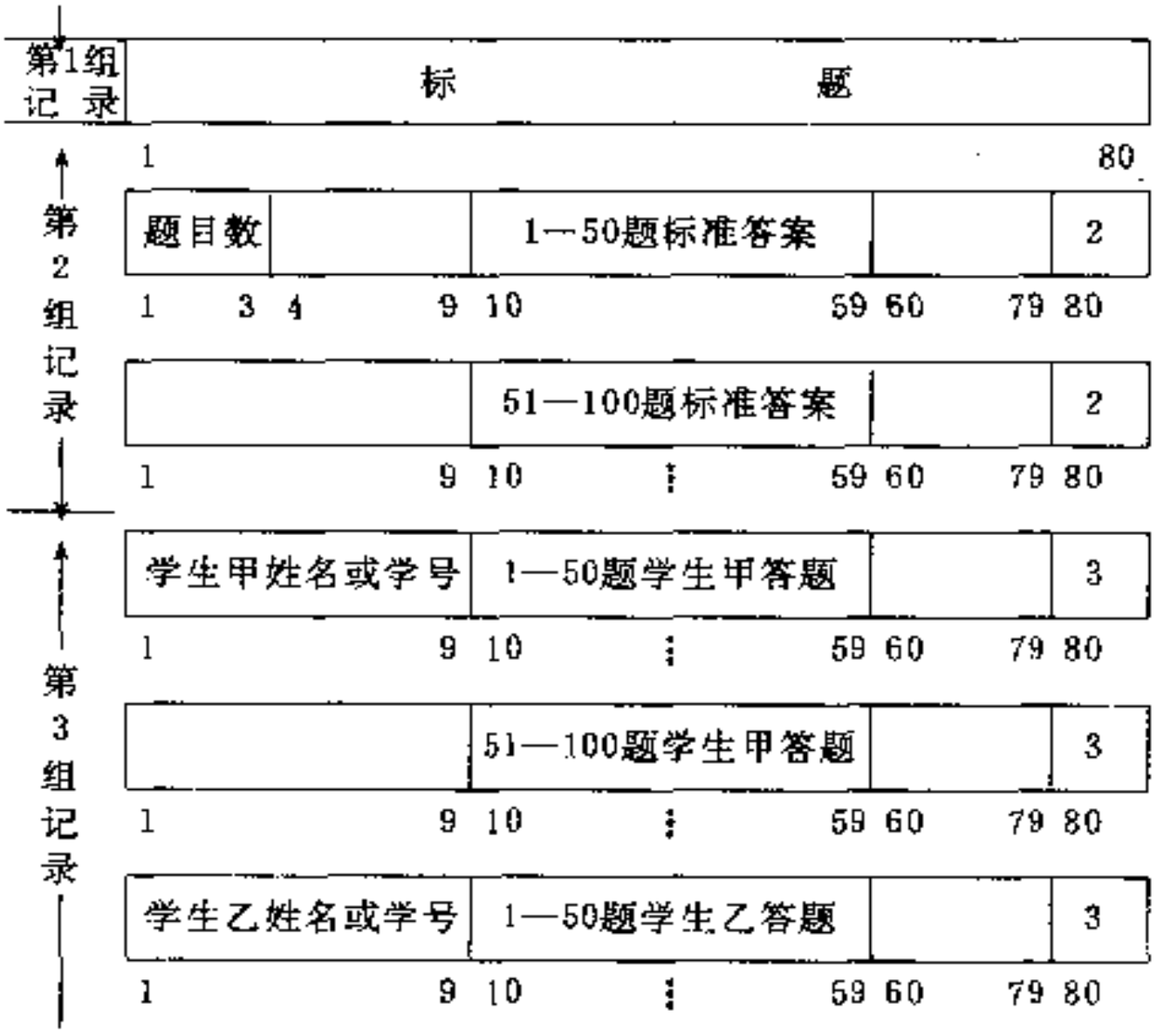


图6.9　学生考卷评分和成绩统计程序输入数据形式

该程序的输出报告有四个：

① 按学生学号排序，每个学生的成绩(答对百分比)和等级报告。

② 按学生得分排序，每个学生的成绩。

③ 平均分数，取最高与最低分之差。

④ 按题号排序，每题学生答对百分比。

以下将分别针对输入条件和输出条件，考虑其边界值设置测

试用例。

输入条件	测　试　用　例
输入文件	空输入文件
标　题	无标题记录 只有一个字符的标题 有 80 个字符的标题
出题个数	出了一个题 出了 50 个题 出了 51 个题 出了 100 个题 出了 999 个题 没有出题 题目数是非数值量
答案记录	标题记录后没有标准答案记录 标准答案记录多一个 标准答案记录少一个
学生人数	学生人数为 0 学生人数为 1 学生人数为 200 学生人数为 201
学生答题	某学生只有一个答卷记录，但有两个标准答案记录 该学生是文件中的第一个学生 该学生是文件中最后一个学生

续表

学生答题	某学生有两个答卷记录，但仅有一个标准答案记录 该学生是文件中的第一个学生 该学生是文件中最后一个学生
输出条件	测　试　用　例
学生得分	所有学生得分相同 所有学生得分都不同 一些学生(不是全部)得分相同(检查等级计算) 一个学生得 0 分 一个学生得 100 分
输出报告①,②	一个学生编号最小(检查排序) 一个学生编号最大 学生数恰好使报告印满一页(检查打印) 学生人数使报告一页打印不够，尚多一人
输出报告③	平均值取最大值(所有学生均考满分) 平均值为 0(所有学生都得 0 分) 标准偏差取最大值(一学生 0 分，一学生 100 分) 标准偏差为 0(所有学生得分相同)
输出报告④	所有学生都答对第 1 题 所有学生都答错第 1 题 所有学生都答对最后一题 所有学生都答错最后一题 报告打完一页后，恰剩一题未打 题数恰好使得报告打印在一页上

以上共计 43 个测试用例，其中有些如果不采用边值分析方法很难考虑到，然而，这些确是很容易发生的问题，可见边值分

析仍然是很有效的方法。

(4) 因果图

前面介绍的等价类划分和边值分析方法都没有考虑到输入情况的各种组合。这样虽然各个输入条件可能出错的情况已经看到了，但多个输入情况组合起来可能出错的情况却被忽略了。采用因果图方法能帮助我们按一定步骤，选择一组高效的测试用例。同时，还能为我们指出程序规格说明的描述中存在什么问题。

利用因果图导出测试用例需要经过以下几个步骤：

① 分析程序规格说明的描述中哪些是原因，哪些是结果。原因常常是输入条件或是输入条件的等价类。结果是输出条件。

② 分析程序规格说明的描述中语义的内容，并将其表示成连接各个原因与各个结果的“因果图”。

③ 由于语法或环境的限制，有些原因和结果的组合情况是不可能出现的。为表明这些特定的情况，在因果图上使用特殊的符号标明约束条件。

④ 把因果图转换成判定表。

⑤ 把判定表的每一列写成一个测试情况。

为了对该方法有进一步理解，需对因果图作一说明。在因果图中出现的四个符号分别表示四种关系(参看图 6.10，其中 C_i 表示原因，通常在图的左部，e_i 表示结果，通常在图的右部。C_i 和 e_i 都可取值 0 或 1，0 表示某状态不出现，1 表示某状态出现)。

① 恒等：若 C_1 是 1，则 e_1 也是 1，否则 e_1 为 0。

② 非：若 C_1 是 1，则 e_1 是 0，否则 e_1 为 1。

③ 或：若 C_1 或 C_2 或 C_3 是 1，则 e_1 是 1，否则 e_1 为 0。“或”可有任意个输入。

④ 与：若 C_1 和 C_2 都是 1，则 e_1 为 1，否则 e_1 为 0。“与”也可有任意个输入。

因果图中使用了简单的逻辑符号，以直线联接左右结点。左

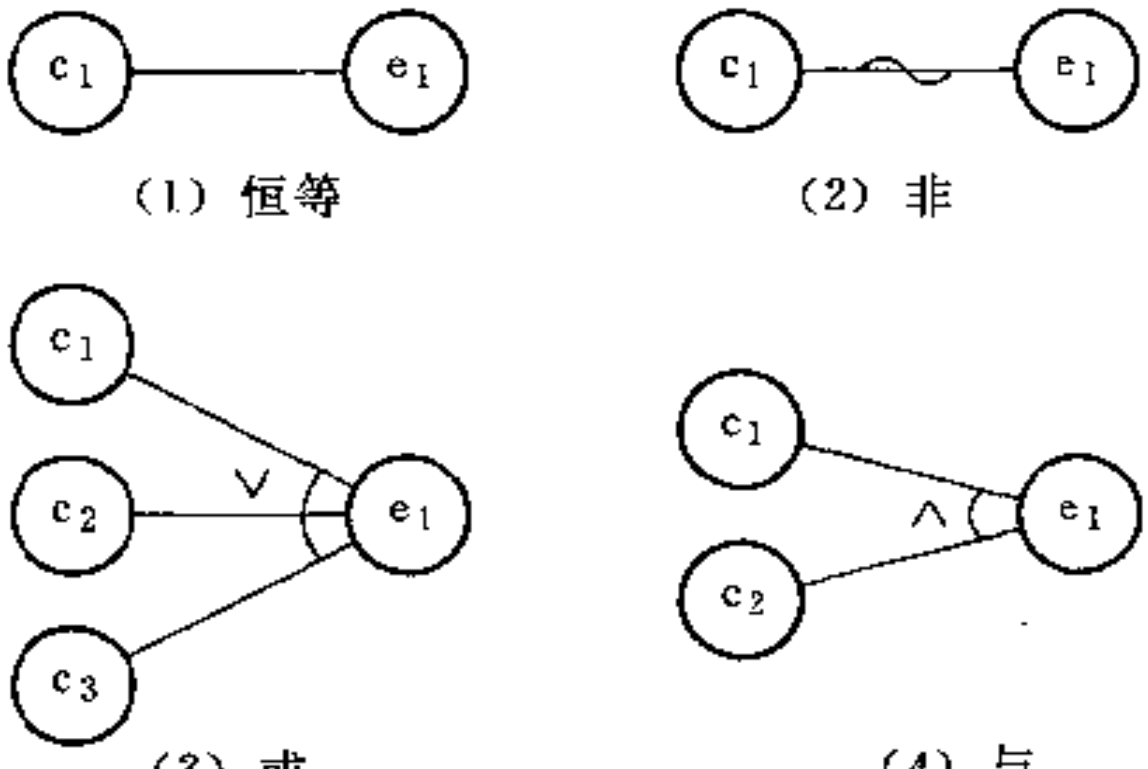


图 6.10　因果图的基本符号

结点表示输入状态(或称原因)，右结点表示输出状态(或称结果)。

我们注意到，在实际问题中，输入状态相互之间还可能存在某些依赖关系，我们称之为“约束”。比如，某些输入条件本身不可能同时出现。输出状态之间也往往存在约束。在因果图中，以特定的符号标明这些约束(参看图 6.11)。

对于输入条件的约束有：

① E 约束(异)：a 和 b 中至多有一个可能为 1，即 a 和 b 不能同时为 1。

② I 约束(或)：a、b 和 c 中至少有一个必须是 1，即 a、b 和 c 不能同时为 0。

③ O 约束(维一)：a 和 b 中必须有一个，且仅有一个为 1。

④ R 约束(要求)：a 是 1 时，b 必须是 1，即不可能 a 是 1 时 b 是 0。

输出条件的约束是 M 约束(强制)：若结果 a 是 1，则结果 b 强制为 0。

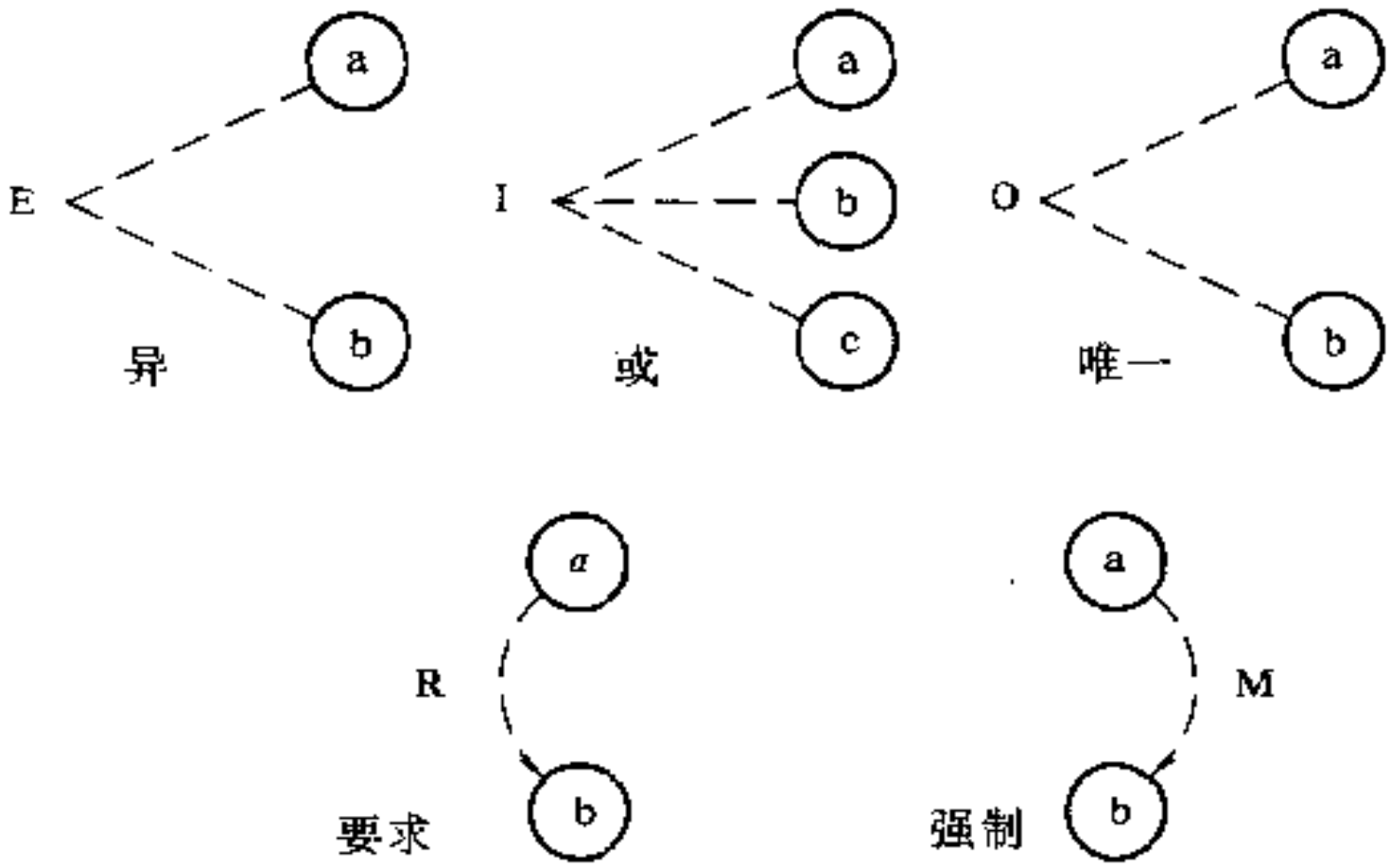


图 6.11　约束符号

现在让我们以一个简单的实例来说明因果图方法。如果某程序的要求是这样规定的：

“第一列字符必须是 A 或 B，第二列字符必须是一个数字，在此情况下进行文件的修改。但如果第一列字符不正确，则给出信息 L。如果第二列字符不是数字，则给出信息 M。”

以下根据这一规格说明画出因果图。在分析以上的要求以后，我们列出原因：

1——第一列字符是 A

2——第一列字符是 B

3——第二列字符是一数字

结果则应是：

21——修改文件

22——给出信息 L

23——给出信息 M

其因果图如图 6.12 所示。图中左列为原因，右列为结果，编号为

11 的中间结点是导出结果的进一步原因。考虑到原因 1 和原因 2 不可能同时为 1，可利用前述的 E 约束，则可得到具有约束的因果图(图 6.13)。

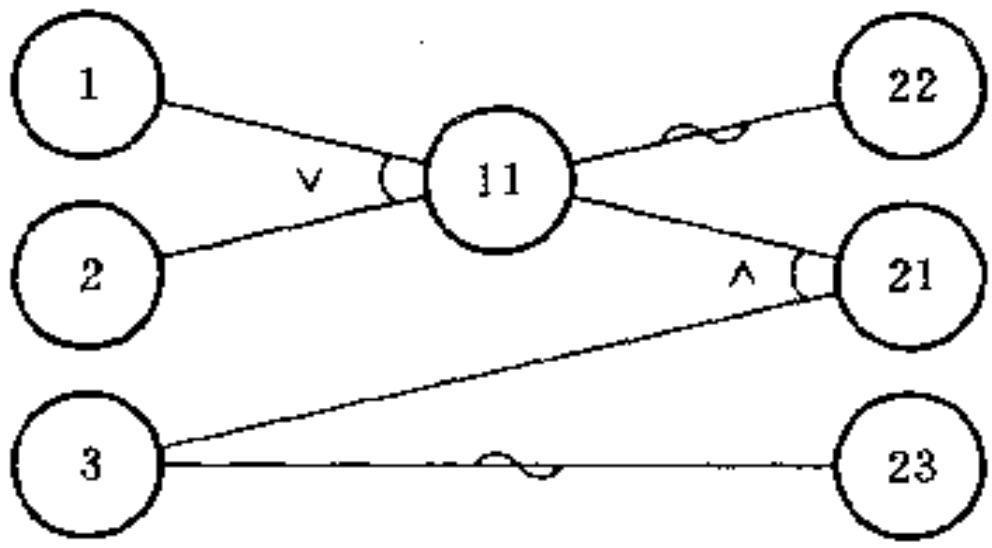


图 6.12　因果图实例

根据因果图建立的判定表是：

		1	2	3	4	5	6	7	8
条件(原因)	①	1	1	1	1	0	0	0	0
	②	1	1	0	0	1	1	0	0
	③	1	0	1	0	1	0	1	0
	⑪			1	1	1	1	0	0
动作(结果)	㉒			0	0	0	0	1	0
	㉑			1	0	1	0	0	0
	㉓			0	1	0	1	0	1
测试用例				A3 A8	AM A?	B5 B4	BN B!	C2 X6	DY P;

注意，表中左面两种情况，原因 1 和原因 2 同时为 1，是不可能出现的。

表的最下一行是针对可能出现的六种情况所给出的测试用例。

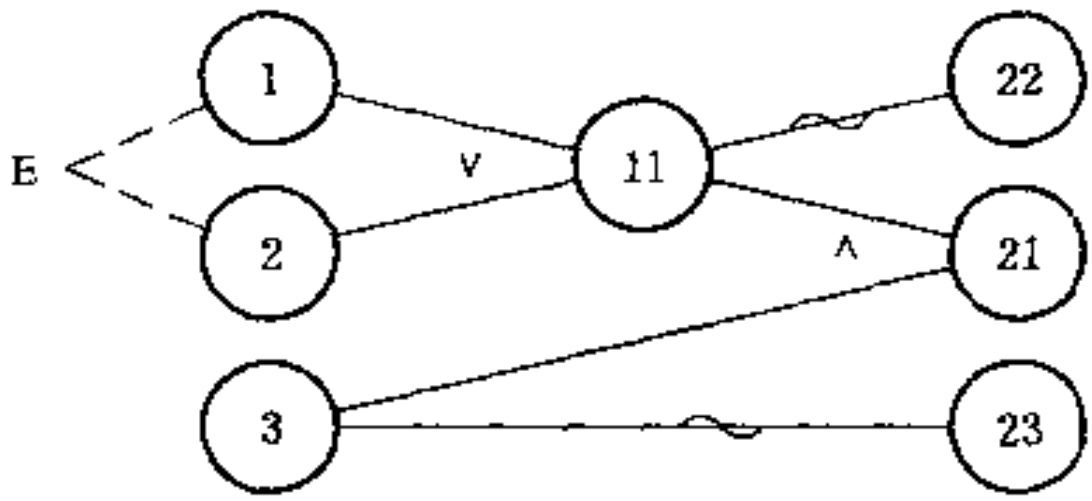


图 6.13　具有 E 约束的因果图

这里只是因果图应用的一个简单实例，限于篇幅，不可能在此举出更复杂的例子。但不要因此而得出结论，似乎因果图和判定表都是多余的。在较为复杂的问题中，它们可以发挥出良好的效果，有力地帮助测试人员确定测试用例。当然，如果某个软件开发项目，在设计阶段就采用了判定表，也就可以不必再画因果图，而是直接从判定表得到测试用例。

6.3　软件测试策略

第一章中曾以瀑布模型描述软件工程过程，为了说明软件测试策略，我们还可把这个过程表达成一个螺旋形(参看图 6.14)。首先，系统工程为软件开发规定了任务，从而把它与硬件要完成的任务明确地划分开。接着便是进行软件需求分析，决定被开发软件的信息域、功能、性能、限制条件并确定该软件项目完成后的确认准则。沿着螺线向内旋转，将进入软件设计和代码编写阶段。从而使得软件开发工作从抽象逐步走向具体化。

软件测试工作也可从这一螺旋线上体现出来。在螺线的核心点针对每个单元的源代码，进行单元测试。在各单元测试完成以后，沿螺线向外前进，开始针对软件整体构造和设计的组装测

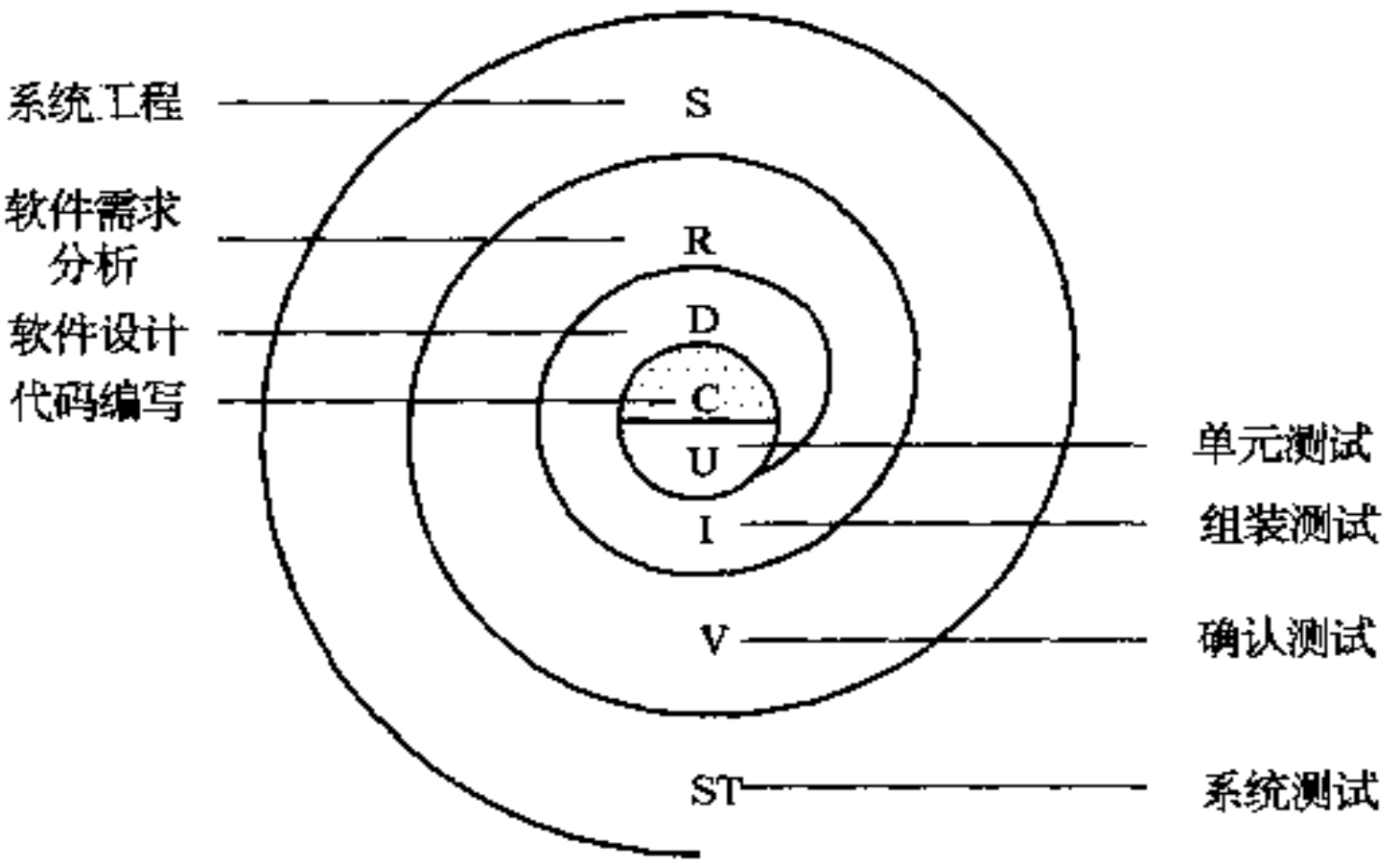


图 6.14　软件测试策略

试。然后是检验软件需求能否得到满足的确认测试，最后，来到螺线的最外层，把软件和系统的其它部分协调起来，当作一个整体，完成系统测试。这样，沿着螺旋线，从内向外，逐步扩展了测试的范围。

以上用螺旋线表明的测试过程，按四个步骤进行，即单元测试、组装测试、确认测试和系统测试。图 6.15 示出测试的四个步

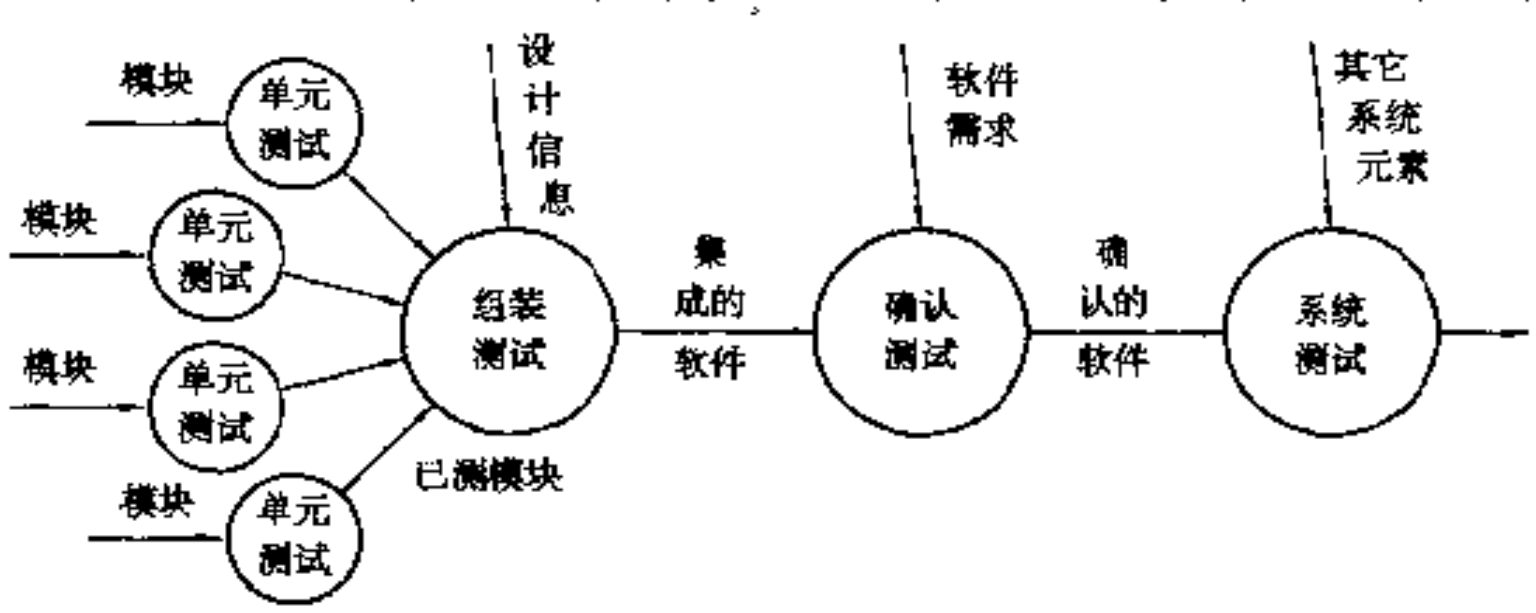


图 6.15　软件测试步骤

骤。开始是分别完成每个单元的测试任务，以确保每个模块能正常工作。单元测试大量地采用了白盒测试方法。尽可能发现模块内部的程序差错。然后，把已测试过的模块组装起来，进行组装测试。其目的在于检验与软件设计相关的程序结构问题。这时较多地采用黑盒测试方法来设计测试用例。完成组装测试以后，要对开发工作初期制定的确认准则进行检验。确认测试是使所开发的软件能否满足所有功能和性能需求的最后保证手段，通常均采用黑盒测试方法。完成确认测试以后，给出的应该是合格的软件产品，但为检验它能否与系统的其它部分(如硬件、数据库及操作人员)协调工作，需要进行系统测试。严格地说，系统测试已超出了软件工程的范围。

以下将围绕上述测试步骤中的有关问题作出进一步说明。

(1) 单元测试

单元测试(unit testing)也称模块测试，这是针对软件设计的最小单位——模块进行正确性检验的测试工作。其目的在于发现各模块内部可能存在的各种差错。单元测试需要从程序的内部结构出发设计测试用例，即采用所谓白盒测试方法。多个模块可以平行地独立进行单元测试：

① 单元测试要解决的问题

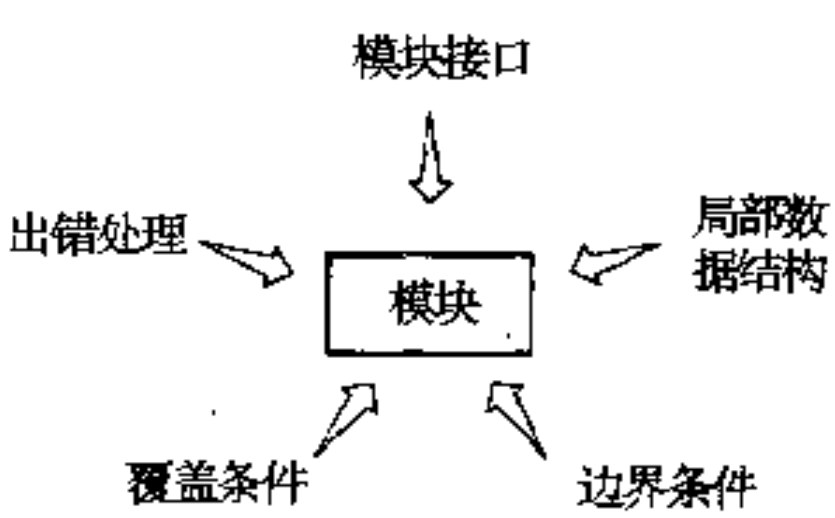


图 6.16　单元测试

单元测试是要针对每个模块的程序，解决以下五个方面的问题(参看图 6.16)：

• 模块接口——对被测的模块，信息能否正常无误地流入和流出。

• 局部数据结构——在模块工作过程中，其内部的数据能否保持完整性，包括内部数

据的内容、形式及相互关系不发生错误。

• 边界条件——在为限制数据加工而设置的边界处，模块是否能够正常工作。

• 覆盖条件——模块的运行能否达到满足特定的逻辑覆盖。

• 出错处理——模块工作中发生了错误，其中的出错处理设施是否有效。

模块与其周围环境的接口有无差错应首先得到检验，否则其内部的各种测试工作也将是徒劳的。Myers 提供的模块接口检查表是很有用的，以下简要地列出：

1）模块接受的输入参数个数与模块的变元个数是否一致？

2）参数与变元的属性是否匹配？

3）参数与变元所使用的单位是否一致？

4）传送给另一被调用模块的变元个数与参数的个数是否相同？

5）传送给另一被调用模块的变元属性与参数的属性是否匹配？

6）传送给另一被调用模块的变元，其单位是否与参数的单位一致？

7）调用内部函数时，变元的个数、属性和次序是否正确？

8）在模块有多个入口的情况下，是否有引用与当前入口无关的参数？

9）是否会修改只是作为输入值的变元？

10）出现全程变量时，这些变量是否在所有引用它们的模块中都有相同的定义？

11）有没有把常数当作变量来传送？

当模块执行了外部的输入、输出时，Myers 提出还需考虑：

1）文件属性是否正确？

2）OPEN 语句是否正确？

3）格式说明与输入、输出语句给出的信息是否一致？

4）缓冲区的大小是否与记录的大小匹配？

5）是否所有的文件在使用前均已打开了？

6）对文件结束条件的判断和处理是否正确？

7）对输入、输出错误的处理是否正确？

8）有没有输出信息的正文错误？

对于局部数据结构应该在单元测试中注意发现以下几类错误：

1）不正确的或不相容的说明。

2）不正确的初始化或省缺值。

3）错误的变量名，如拼写错或缩写错。

4）不相容的数据类型。

5）下溢、上溢或是地址错误。

除局部数据结构外，在单元测试中还应弄清楚全程数据（如FORTRAN 的 COMMON)对模块的影响。

如何设计测试用例，使得模块测试能够高效率地发现其中的错误，这是非常关键的问题。无论考虑何种逻辑覆盖都应注意发现以下一些典型的计算错误：

1）对运算优先性的错误理解，或是错误的处理。

2）运算方式(mode)未加区分，发生了混合运算的情况。例如，实型量和复型量混淆。

3）初始化错误。

4）计算精确度不够。

5）表达式中符号表示的错误。比较和控制流常常是彼此密切相关的，比较的错误势必导致控制流的错误。

需要特别注意发现的错误包括：

1）不同数据类型数据进行比较。

2）逻辑运算符或其优先级用错。

3）本应相等的数据，由于精确度原因而不相等。

4）变量本身或是比较有错。

5）循环终止不正确，或循环不已。

6）在遇到发散的循环时，不能摆脱出来。

7）循环控制变量修改有错。

程序运行中出现了异常现象并不奇怪，良好的设计应该预先估计到，将来投入运行后可能发生什么出错的情况，并给出相应的处理措施，使得用户不致于发生了这些情况束手无策。检验程序中出错处理这一问题解决得怎样，可能出现的情况有：

1）对运行发生的错误描述得难于理解。

2）指明的错误并非实际遇到的错误。

3）出错后，尚未进行出错处理便引入系统干预。

4）意外处理不当。

5）提供的错误信息不足，以致无法找到出错的原因。

边界测试是单元测试的最后一步，是不容忽视的。实践表明，软件常常在边界地区发生问题。例如，处理 n 维数组的第 n 个元素时很容易出错，循环执行到最后一次执行循环体时也可能出错。这可利用边值分析方法来设计测试用例，以便发现这类程序错误。

② 单元测试的步骤

单元测试常常被当作代码编写的附属步骤，也有人把代码编写和单元测试当作一个开发阶段考虑。显然，它是在程序编写完毕了、经过复查、确认没有语法错误以后，针对每个程序模块单独进行的测试工作。

由于每个模块在整个软件中并不是孤立的，我们在对每个模块进行单元测试时，也不能完全忽视它们和周围模块的相互联系。为模拟这一联系，在进行单元测试时，需设置若干辅助测试模块。辅助模块有两种，一种是驱动模块(driver)，用以模拟被测

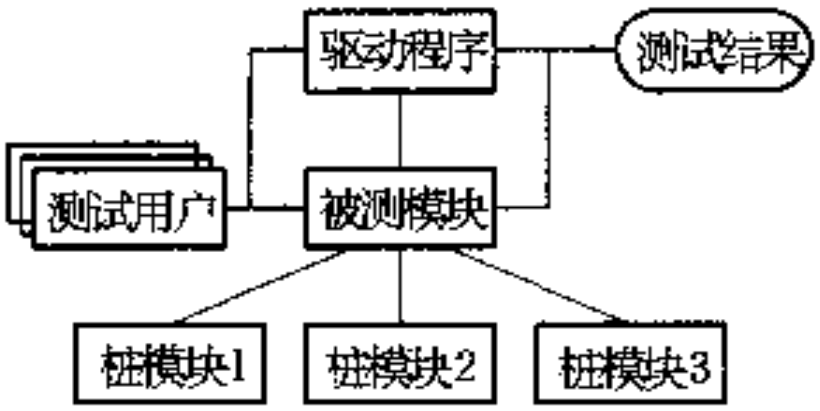


图 6.17　单元测试环境

模块的上级模块；另一种桩模块(stub)，用以模拟被测模块工作过程中所调用的模块。图 6.17 示出一个被测模块进行单元测试时的环境状况。其中设置了一个驱动模块和三个桩模块。驱动模块在单元测试中接受测试数据，把相关的数据传送给被测模块，启动被测模块，并打印出相应的结果。桩模块由被测模块调用，它们仅做很少的数据处理，例如打印入口和返回，便于检验被测模块与其下级模块的接口。

自然，驱动模块和桩模块对测试人员来说是一种额外的负担，就是说，虽然在单元测试中必须编写这些辅助模块的程序，但却不作为最终的软件产品提供给用户。好在这些模块的结构十分简单，模块间接口的全面检验可在组装测试时去进行。

(2) 组装测试

在每个模块完成单元测试以后，需要按照设计时作出的结构图，把它们联结起来，进行组装测试(integrated testing)。经验不多的人可能会提出，既然在单元测试时已对所有模块的工作是否正常进行了检验，为什么还要联起来再测试呢？实践表明，一些模块能够单独地正常工作，并不能保证联结起来也能正常工作，程序在某些局部反映不出的问题，在全局上很可能暴露出来，影响功能的发挥。

怎样合理地组织组装测试，这里提供两种不同的方法，即非增式测试和增式测试。

非增式测试方法是这样进行的：在配备辅助模块的条件下，对所有模块进行个别的单元测试。然后在此基础上，按程序结构图将各模块联结起来，把联结后的程序当作一个整体进行测试。

图 6.18 表示采用这种非增式的组装测试方法的一个例子。被测程序的结构图为子图(a)，由六个模块构成。在进行单元测试时，根据它们在结构图中的地位，对模块 B 和 C 配备了驱动模块和桩模块，对模块 D、E 和 F 只配备了驱动模块。对主模块 A，由于它处在结构图的顶端，再没有其它模块调用它，因此只为它配备了三个桩模块，以模拟被它调用的三个模块 B、C 和 D。分别进行单元测试以后，再按(a)的结构图形式联结起来，进行组装测试。

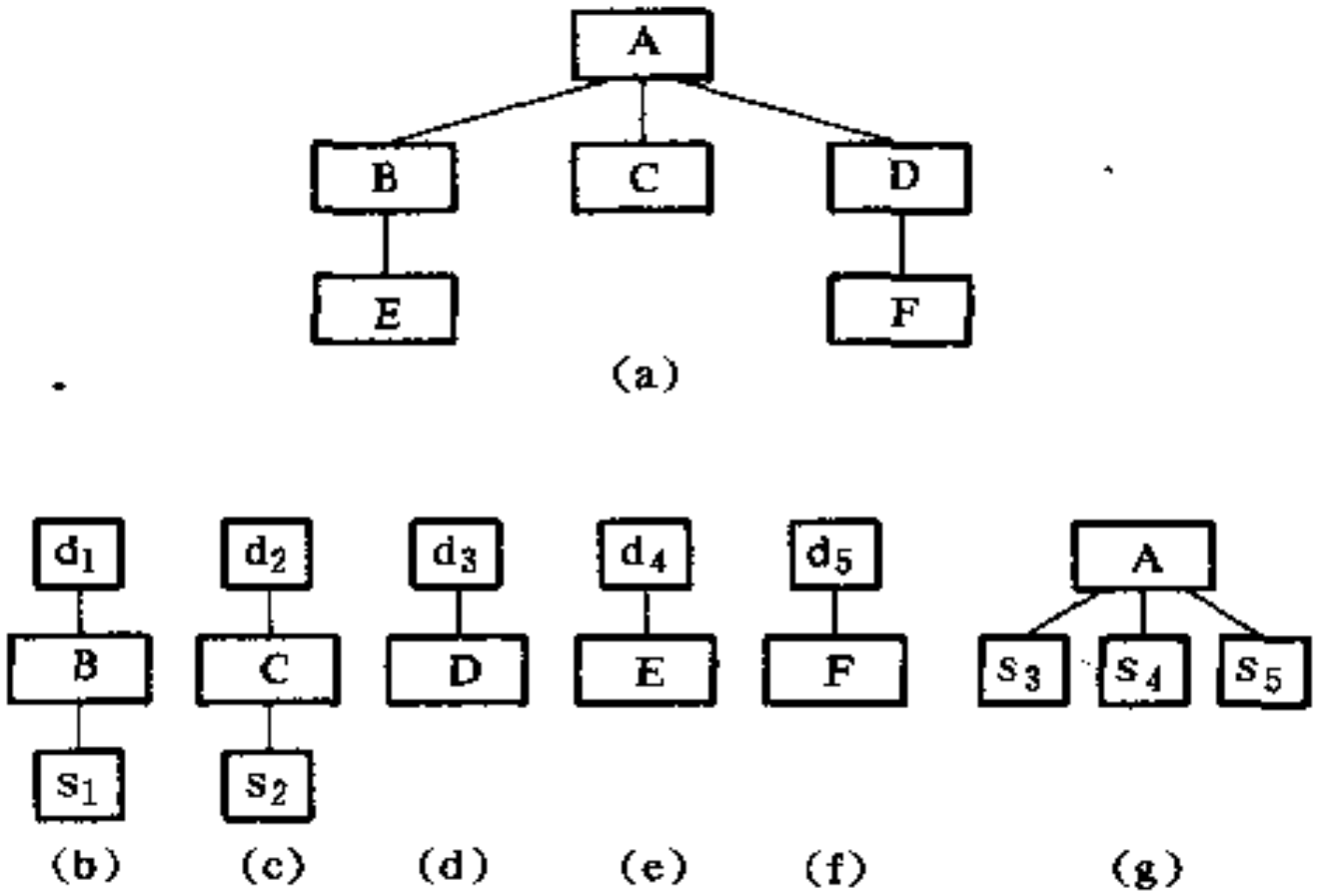


图 6.18　非增式测试例

增式测试的作法与非增式测试有所不同。它的集成是逐步实现的，组装测试也是逐步完成的。也可以说它把单元测试与组装测试结合起来，一道进行，增式组装测试可按不同的次序实施，因而可以有两种：

自顶向下增式测试表示逐步集成和逐步测试是按结构图自上而下进行的。图 6.19 给出的程序，其结构图如子图(c)所示。组装测试分为三步：首先，对顶层的主模块 A 进行单元测试，这时需配以桩模块 S_1、S_2 和 S_3(参看子图(a))，以模拟被它调用的模块 B、C

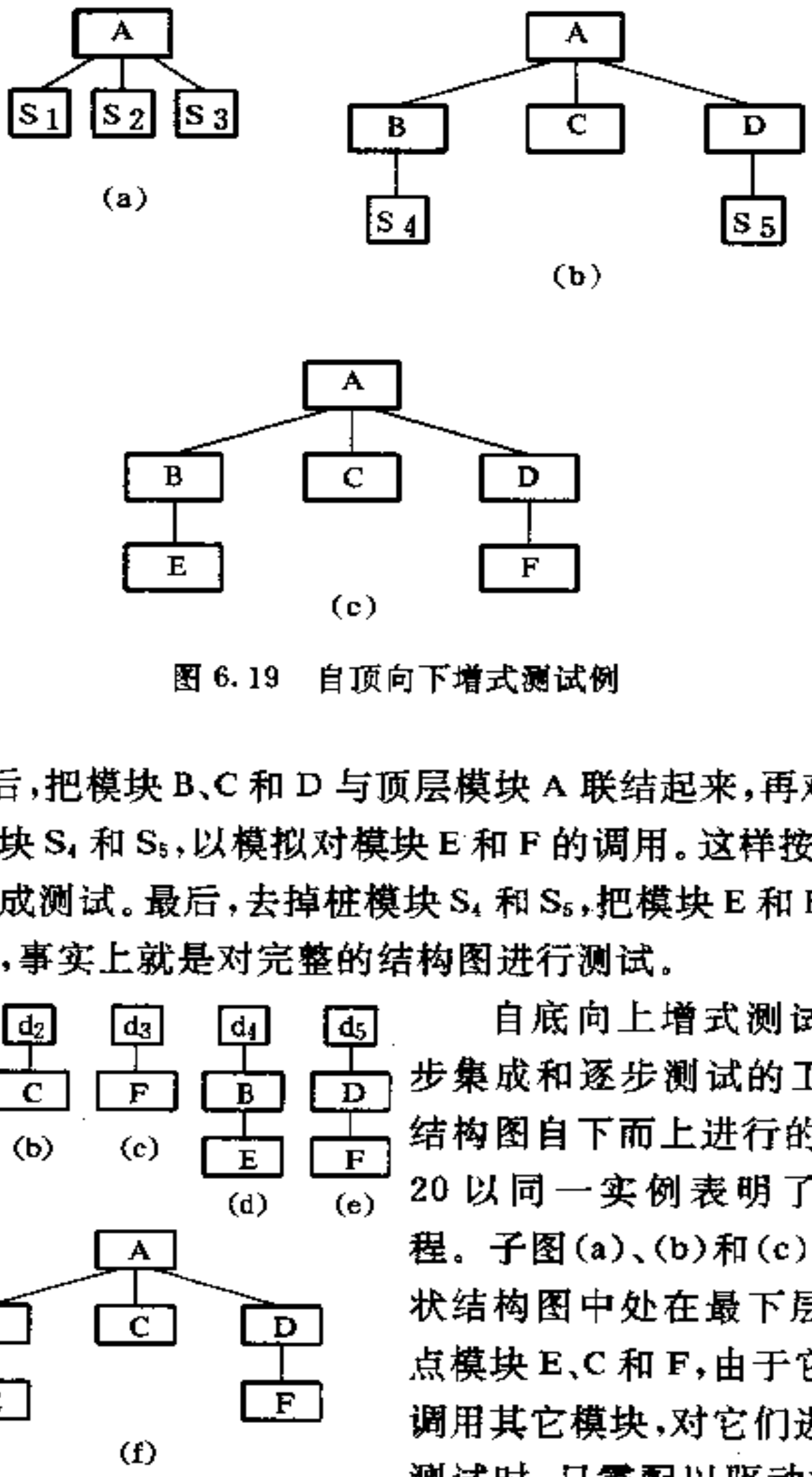


图 6.19　自顶向下增式测试例

和 D。其后，把模块 B、C 和 D 与顶层模块 A 联结起来，再对 B 和 D 配以桩模块 S_4 和 S_5，以模拟对模块 E 和 F 的调用。这样按子图(b)的形式完成测试。最后，去掉桩模块 S_4 和 S_5，把模块 E 和 F 联上去进行测试，事实上就是对完整的结构图进行测试。

图 6.20　自底向上增式测式例

自底向上增式测试表示逐步集成和逐步测试的工作是按结构图自下而上进行的。图 6.20 以同一实例表明了这一过程。子图(a)、(b)和(c)表示：树状结构图中处在最下层的叶结点模块 E、C 和 F，由于它们不再调用其它模块，对它们进行单元测试时，只需配以驱动模块 d_1、d_2 和 d_3，用来模拟 B、A 和 D 对

它们的调用。完成这三个单元测试以后，再按子图(d)和(e)的形式，分别将模块 B 和 E 及模块 D 和 F 联结起来，在配以驱动模块 d_4 和 d_5 的条件下实施部分组装测试。最后再按子图(f)的形式完成整体的组装测试。

比较以上几种组装测试的做法，我们可以看出，非增式测试的做法是先分散测试，再集中起来一次完成组装测试。如果在模块的接口处存在差错，只会在最后的集成时一子下暴露出来。与此相反，增式测试的逐步集成、逐步测试的办法，把可能出现的差错分散暴露出来。这就为找出问题和修改带来了方便。其次，从前面的实例中也能看出，增式测试使用了较少的辅助模块，也就减少了辅助性测试工作。并且一些模块在逐步集成的测试中，得到了较为频繁的考验，因而可能取得较好的测试效果。总的说来，增式测试比起非增式测试来，具有一定的优越性。

(3)确认测试

组装测试以后，分散开发的模块被联接起来，构成完整的软件包。其中各模块间接口存在的种种问题都已消除。于是测试工作进入最后阶段——确认测试(Validation testing)。什么是确认测试，有着多种不同的说法。其中最简明，也是最严格的解释是检验所开发的软件，看它是不能按顾客提出的要求运行。若能达到这一要求，便认为开发的软件是合格的。因而有的软件开发部门把确认测试称为合格性测试(qualification testing)。这里所说的顾客要求通常指的是在软件规格说明书中确定的软件技术指标，或是专门为测试所规定的确认准则。

① 确认测试准则

怎样来判断被开发的软件是成功的?为了确认它的功能、性能以及限制条件是否达到了要求，应进行怎样的测试?在需求规格说明书中可能作了原则性规定，但在测试阶段需要更详细、更具体地在测试规格说明书中作进一步说明。例如，制定测试计划时，要说

明确认测试应测试哪些方面，并给出必要的测试用例。除了考虑功能、性能以外，还需检验其它方面的要求，例如，可移植性、兼容性、可维护性、人机接口以及开发的文件资料等是否符合要求。

经过确认测试，应该为已开发的软件作出结论性评价。这也无非是两种情况之中的一个：(1)经过检验的软件功能、性能及其它要求均已满足需求规格说明书的规定，因而可被接受。认为是合格的软件。(2)经过检验发现与需求规格说明书的规定有相当的偏离，得到一个各项缺陷清单。对于第二种情况，往往很难在交付期以前把发现的问题纠正过来。这就需要开发部门和顾客进行协商，找出解决的办法。

② 配置审查

配置审查是确认过程的重要环节。其目的在于确保已开发软件的所有文件资料均已编写齐全，并得到分类编目，足以支持投入运行以后的软件维护工作。这些文件资料包括：用户所需资料(如用户手册、操作手册)，设计资料(如设计说明书等)，源程序以及测试资料(如测试说明书、测试报告等)。配置审查(configuration review)有时也称配置审计(configuration audit)。图 6.21 给出了它和确认测试的关系。

③ 验收测试

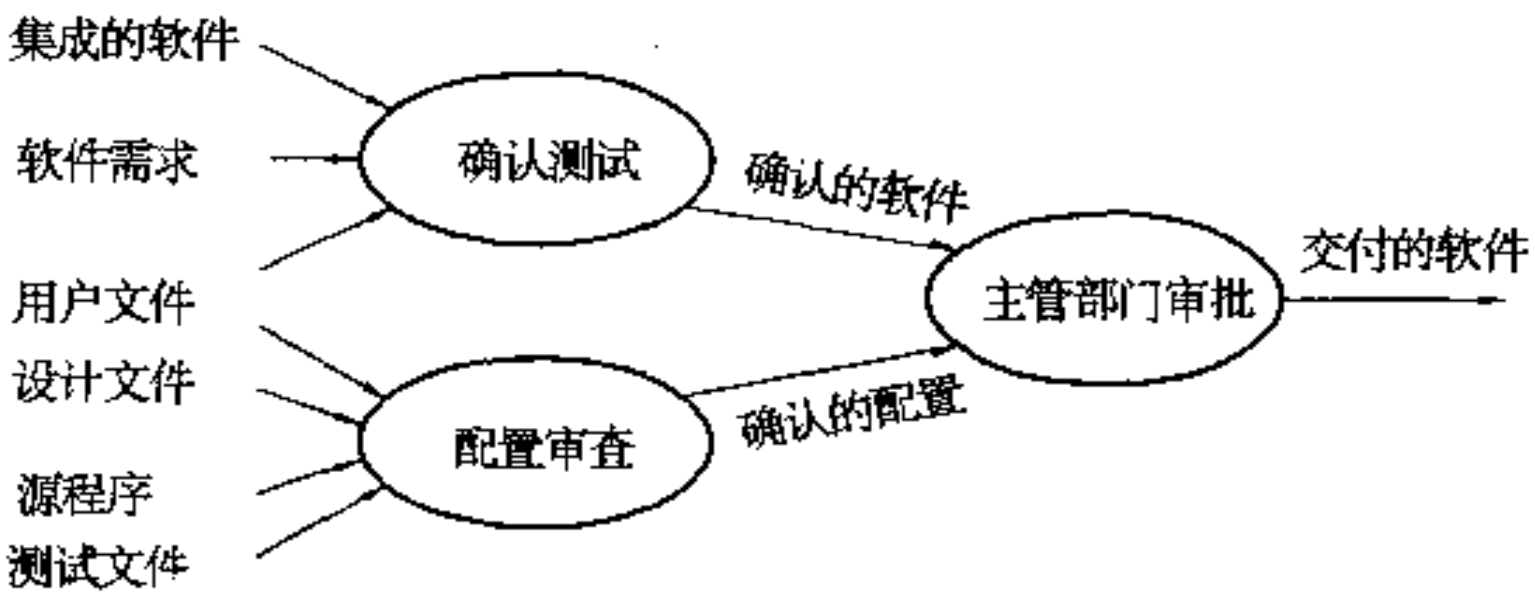


图 6.21　配置审查与确认测试

如果开发的软件是应某个客户的请求而订制的情况，就要通过验收测试(acceptance testing)来确认客户所提出的需求是否已得到全面的满足。我们曾把这样的软件称为项目软件(参看第一章有关软件分类的部分)。

验收测试是为料确认已开发的软件能否达到验收标准，客户决定是否接受的正式测试过程。其验收标准应能体现软件的功能、性能及其它一些特性。所谓正式测试是指，这种测试不只考虑软件某个方面的质量，而是对软件质量的全面检验。除涉及到源程序外，还会涉及到编写的文件资料和软件使用特性。如果各方面需求都得到满足，则可通过验收，已开发的软件便可以交付客户。

由于验收测试关系到已开发软件的命运，其测试过程和测试结果应该是客观的，并且符合实际。比如，不应受到个人因素的影响，而偏向于开发者或客户一方的利益。还应注意，验收测试应在软件投入运行后所处的实际生产环境下进行，而不应在软件开发环境下进行，以利于去掉一些不切实际的人为模拟条件。

既然要检验客户需求被满足的情况，软件的验收测试就理应由客户主持进行。在特殊的情况下，客户也可委托第三方实施或邀请第三方参与，其目的在于排除开发者一方片面观点的影响。

制定验收标准是验收测试的关键。为此要针对软件项目的具体情况，制定详细验收测试计划，并在其中明确规定验收标准。验收测试中发现源程序运行中有错误，也并不是十分罕见的现象。问题在于恰当地掌握验收通过的标准。例如，测试中运行五千行可执行语句的源程序不得出现两个以上的程序差错，或者限制在指定的运行时间内，如 30 分钟，不得出现两个程序差错等。

(4)系统测试

由于软件只是基于计算机的数据处理系统中的一个组成部分，软件开发完成以后，最终还要与系统中的其它部分配套运行。系统在投入运行以前各部分需完成组装和确认测试，以保证各组

成部分不仅能单独地受到检验，而且在系统各部分协调工作的环境下也能正常工作。系统的确认测试自然已经超出了软件工程的范围。然而，软件在系统中毕竟占有相当重要的位置，软件的质量如何，软件的测试工作进行得是否扎实，势必与能否顺利、成功地完成系统测试关系极大。另一方面，系统测试实际上是针对系统中各个组成部分进行的综合性检验。尽管每一个检验有着特定的目标，然而所有的检测工作都要验证系统中每个部分均已得到正确的集成，并能完成指定的功能。以下分别简要地说明几种系统测试：

① 恢复测试

恢复测试是要采取各种人工干预方式使软件出错，而不能正常工作，进而检验系统的恢复能力。如果系统本身能够自动地进行恢复，则应检验：重新初始化、检验点设置机构、数据恢复以及重新启动是否正确。如果这一恢复需要人为干预，则应考虑平均修复时间是否在限定的范围以内。

② 安全测试

安全测试的目的在于验证安装在系统内的保护机构事实上能够对系统进行保护，使之不受各种非常的干扰。系统的安全测试需要设置一些测试用例试图突破系统的安全保密措施，检验系统是否有安全保密的漏洞。

③ 强度测试

检验系统的能力最高能够达到什么实际限度。进行强度测试时，让系统的运行处于资源的异常数量、异常频率和异常批量的条件下。例如，如果正常的中断平均频率为每秒一到二次，强度测试设计为每秒 10 次中断。又如某系统正常运行可支持 10 个终端并行工作，强度测试则检验支持 15 个终端并行工作的情况。

④ 性能测试

性能测试检验安装在系统内的软件运行性能。这种测试常常

与强度测试结合起来进行。为记录性能需要在系统中安装必要的量测仪表或是为了度量性能而设置的软件(或程序段)。

(5) 人工测试

本章前面介绍的方法，基本上是借助计算机辅助进行测试的方法。这些方法在提高测试效率方面具有它的优越性。但我们不应因此而忽视另外一类，所谓不依赖于计算机的测试方法，也称之为人工测试。实践表明，人工测试也能相当有效地查出错误。据美国 IBM 公司的统计，采用人工测试的方法找出的错误占所有发现错误的 80%。在一般的软件开发项目中试用一种或几种人工测试技术肯定是十分有益的，并且它和利用计算机进行的测试并不矛盾，而是互为补充的。

在代码编写完、尚未进行计算机测试之前，使用人工测试，可以作为上机测试的准备，消除比较容易发现的程序差错。在设计阶段完成以后，也可采用人工检查的办法发现问题。其实，人工测试的做法绝不只限于此，它完全可以运用到软件开发的各个阶段，并且被证明是十分有效的。

人工测试有着许多不同的形式。比如，以逐行逐句审查源程序代码为目标的，由程序开发者主持，聘请其它程序员或分析员参加的程序审查会(Code Reading Review)。以审查某些开发阶段成果为目标，由参加人员(可能包括程序开发人员、测试人员、用户，甚至维护人员)轮流主持评审的活动(Round-Robin Review)。与程序审查会类似，但要求与会者按指定的数据运行被审查程序的程序遍查会(Walkthroughs)。还有由个人阅读程序，依照查错表来检查程序或用测试数据按程序“走”一遍的 静 态 检 查(Desk Ckecking)等等。

下面以最为普遍使用的开发阶段评审(inspection)为例作一简要介绍。

评审会举行以前自然已经确定了评审的目标和目的。例如针

对软件设计或是源代码的审查。首先起草评审文件(Inspection Schedule Memo)，其中规定评审的要求、参加人员的责任、待审查的项目和评审进度等。评审会前做好准备工作，包括把被审查资料提前送交与会者，除被审查的软件产品资料外，还应提供标准或规范资料及检查项目清单(checklist)。标准或规范是评审工作的依据。检查项目清单则详细列举了各个检查项的特性，类似于本章单元测试中给出的模块接口检查表。提前送交的目的是使与会者有时间事先阅读或作初步的分析和研究。

评审会的参加者应包括被审查软件的开发人员、用户代表、维护人员代表及标准规范实施的检查人员。管理人员通常不参与评审，是为了避免人们把对软件产品的的质量研讨理解成为对开发人员个人工作成绩的考核和评价。

评审会应首先由开发人员提出报告，对审查的项目作出详细的说明。也可以使用检查项目清单逐项进行，以便做到全面无遗漏地检查。与会者在会上可提出各种问题，要求开发人员给出解答，同时也可提出评论性意见或建议。评审会设有专人负责记录，特别应该把发现的问题或差错以及评论或建议逐条详细地记录下来。评审会的时间不宜过长，通常超过两小时便会降低评审的效率，或是可能放过一些问题未被查出。评审会最后做出通过与否的结论性意见并附有与会者的签名。

评审会后根据记录整理出评审会简报，其中应扼要地反映评审会发现的问题，作为向管理部门的报告。评审会后自然要对发现的问题和错误作出处理或修正，必要时向管理部门报告改正错误所需的时间和耗费的资源。典型的错误应用以扩充检查项目清单，以便再作评审时注意到这些问题。

第七章　软件维护

软件维护是指已完成开发工作，交付使用以后，对软件产品所进行的一些软件工程活动。一个中等规模的软件，如果开发阶段需要一年到二年的时间，在它投入使用以后，其运行或工作时间可能持续五年到十年。那么它的维护阶段正是运行的五年到十年期间。在这段期间内对软件进行维护，人们几乎需要着手解决开发阶段所遇到的各种问题，同时还要解决某些维护工作本身的特有问题。做好软件的维护工作不仅能排除障碍，使它能够正常工作，还可以使它扩充功能，提高性能，为用户带来显著的效益。可惜的是，直到目前为止，人们对软件维护的认识远远不如软件开发。因为，开发工作更容易被看到，容易被人们重视。有人曾经以浮在海面上的大冰块作比喻，露在水面上的部分，如同软件开发阶段的工作，被人们一眼看到，殊不知浸在水下的部分看不到，可实际上它的体积要大得多。和软件开发工作对比，软件维护的工作量和成本都要大得多。近年来，人们透过水面正逐渐看清水下的冰块，原来对维护工作所抱有的那种片面认识正在逐步改变。

7.1　什么是软件维护

（1）软件维护工作是必要的吗？

在软件开发阶段结束以后，作为产品的软件交付用户使用，进入运行阶段。实践表明，在运行中仍然有必要对软件进行更动，原因有以下多个方面：

① 改正在运行中新发现的软件错误和设计上的缺陷。这些错误和缺陷是在开发后期测试阶段未能发现的。

② 改进设计，以便增强软件的功能，并提高软件的性能。

③ 要求已运行的软件能适应特定的硬件、软件、外部设备、通信设备的工作环境或是要求适应已变动的数据或文件。

④ 为使投入运行的软件与其它相关的程序有良好的接口，以利于协同工作。

⑤ 为使运行的软件的应用范围得到必要的扩充。

需要注意的是，这里所说的软件"维护"是从硬件的"维护"一词借用过来的。其实两者的含意有着很大的不同。对硬件来说"维护"意味着"维修"，它表示更换已损坏的零部件，或是对机械零件进行擦洗、注油，加以润滑保养。然而，这并不会影响到设备的功能，对设备性能的提高也是非常有限的。也许对提高设备工作的效率和延长设备的使用期限是有作用的。软件的维护工作则完全是另外一回事。软件的维护不仅可以改正原来设计中的错误或不当之处，而且还能增强软件的功能，提高它的性能。

怎样正确看待软件维护工作，是做好软件维护的重要前提。我们知道，软件维护处在软件生命期的最后阶段。在此以前完成的开发工作曾经花去了大量的人力和资源。开发完成以后，用户迫切希望它能正常工作，发挥效益。同时也希望它能稳定可靠地工作，具有较长的使用寿命。然而，实践表明，任何一个软件在通过验收测试以后，谁都不能保证，软件内部所有的隐藏错误完全排除了。随着对它的频繁使用。某些原来隐蔽的问题会逐渐暴露出来。用户还会发现一些使用不便之处。面对这些问题，为了让它能正常有效地工作，有必要再次投入一定数量 的人力和资源，开展软件维护工作。统计资料表明，维护阶段的花费占整个软件生命期花费的 67%。这是一个相当可观的数字。如果我们不能充分认识到维护工作的重要性和迫切性，在技术上、人力安排上和投资上给予足够的重视，势必使已开发的软件无法发挥应有的效益。直到 70 年代后期，人们的主要注意力一直集中在开发阶段，着重研究程序设

计方法及其工具，忽视软件的维护。近年来人们开始认识到维护现有软件的重要意义，重视起维护工作来。图 7.1 表明了软件开发和维护成本比例的变化。

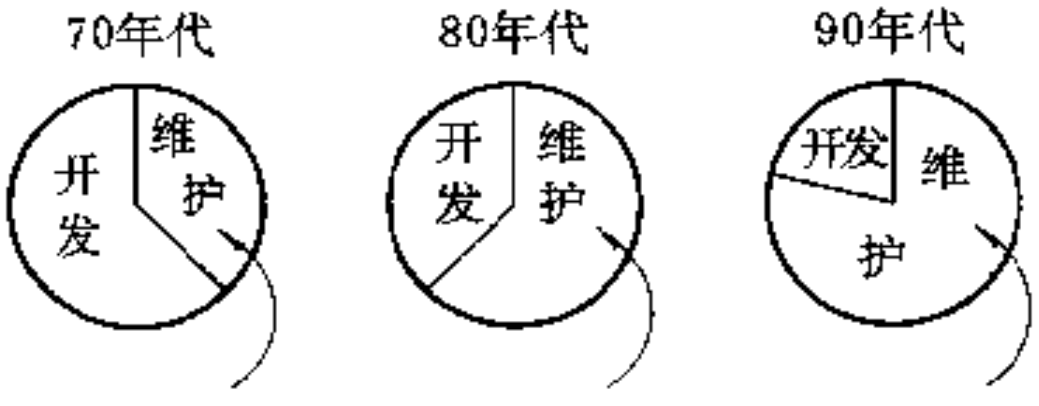


图 7.1　维护与开发的成本对比

我国软件工程的实践还很不够，许多人对于维护工作至今尚未充分地理解。有人甚至错误地否认维护工作的必要性。不少用户对软件维护的知识了解得很少，一些重要的软件运行过程中，没有专人负责维护。致使有的软件在运行中出现了问题，因得不到解决而被放置起来，遭受经济损失。也有人以为，运行中发生的问题都应该由原软件开发人员或原开发部门来解决。其实，他们并不理解，已交付的软件产品一旦投入运行，并且超过合同规定的保修期以后，开发人员和开发部门就完全没有责任了。否则，软件开发人员就会被大量的维护工作所束缚。开发的软件越多，承受维护工作的负担也越重，最终将导致完全没有时间和精力从事新的软件开发工作。很显然，这种情况对软件产业的发展是极为不利的。

(2) 软件维护工作的内容

通常认为，软件的维护工作包括以下三个方面：

- 改正性维护(Corrective maintenance)
- 适应性维护(Adaptive maintenance)
- 完善性维护(Perfective maintenance)

改正性维护是在软件运行中发生异常或故障时进行的。这种

故障常常是由于遇到了从未用过的输入数据组合情况或是发生在与其它软件的接口或与硬件的接口出现了问题。严重的故障未能及时解决，势必使得它所支持的数据处理活动被迫停止。究其原因，这些故障是由于软件开发过程中某个环节上的隐错造成的。在开发的末期所进行的测试未能将其发现，带着这些隐错运行，只是在某些特定情况下才暴露出来。有人统计分析后得知，在典型的市售软件包中，含有缺陷的代码行占代码总行数的千分之三。事实上，即使运行多年的软件在某种特定的情况下，仍然可能暴露出开发中隐藏的问题来，这是不足为奇的。然而对已发现的问题进行修改，一般都应十分谨慎。在制订修改计划后还需经过复审，修改工作在严密的控制下进行，以防造成不良后果。

改正性维护的例子有：

- 改正原来程序中未使开关复原的错误
- 解决开发时未能测试各种可能条件带来的问题
- 解决原来程序中遗漏处理文件中最后一个记录的问题

适应性维护是要使运行的软件能适应外部环境的变动。我们知道，计算机技术近年来发展得越来越快，几乎每三年就要出现一代新的计算机硬件。另一方面，新的操作系统和原来操作系统的新版本不断涌现。建立在硬件和操作系统上的应用软件，其实用年限常常不止三年，长者超过十年。这便要求应用软件能跟上发展的新形势，使之不致因不能适应操作系统的新版本而影响正常工作。除此以外，“数据环境”的变动也要求进行适应性维护。例如，数据库的变动、数据格式的变动、数据输入输出方式的变动以及数据存储介质的变动等都会直接影响到软件的正常工作。

适应性维护的例子有：

- 为现有的某个应用问题实现一个数据库管理系统。
- 对某个指定编码进行修改，从三个字符改成四个字符。

• 缩短系统的应答时间，使其达到特定要求。

• 调整两个程序，使其可以使用相同的记录结构。

• 修改程序，使其适用于另外的终端。

完善性维护则是为扩充软件的功能、提高原有软件性能而开展的软件工程活动。这里所说的新功能和新性能都是在原来开发中编制的软件需求规格说明书上并未规定的内容。用户在使用了一段时间以后，提出了新的要求，希望在原已开发软件的基础上加以扩充。例如，

• 修改计算工资程序，使其增加新的扣除项目。

• 在已有的性能分析程序中增加包含若干属性的新报告。

• 把现有程序的终端对话方式加以改造，使其具有方便用户使用的界面。

• 改进图形输出。

• 增加联机求助(HELP)命令。

• 为软件的运行增加监控设施。

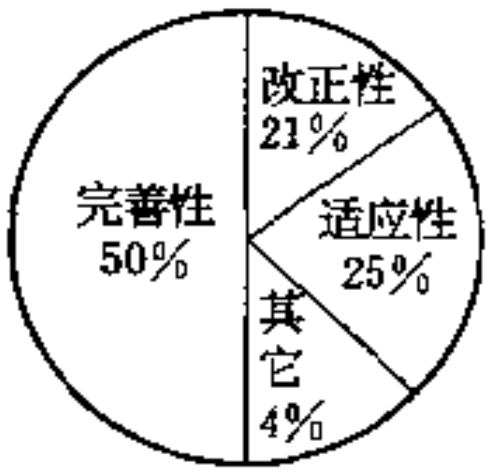


图7.2 几类软件维护所占比例

以上三种类型的维护工作在实践中占有不同的比例(请参看图 7.2)。其中完善性维护工作占的比例最大。

(3) 维护工作的数据流

维护工作的过程可从图 7.3 给出的维护数据流中看到。其实，除了增加维护的管理环节以外，其它部分工作和软件开发的过程极为相似。

7.2 维护工作存在的问题及其分析

(1) 当前维护工作存在哪些问题

当前软件维护工作仍然存在着一些实际问题，其中有些是属于认识上的，有些是客观存在的。不过，问题的解决不仅仅需要维

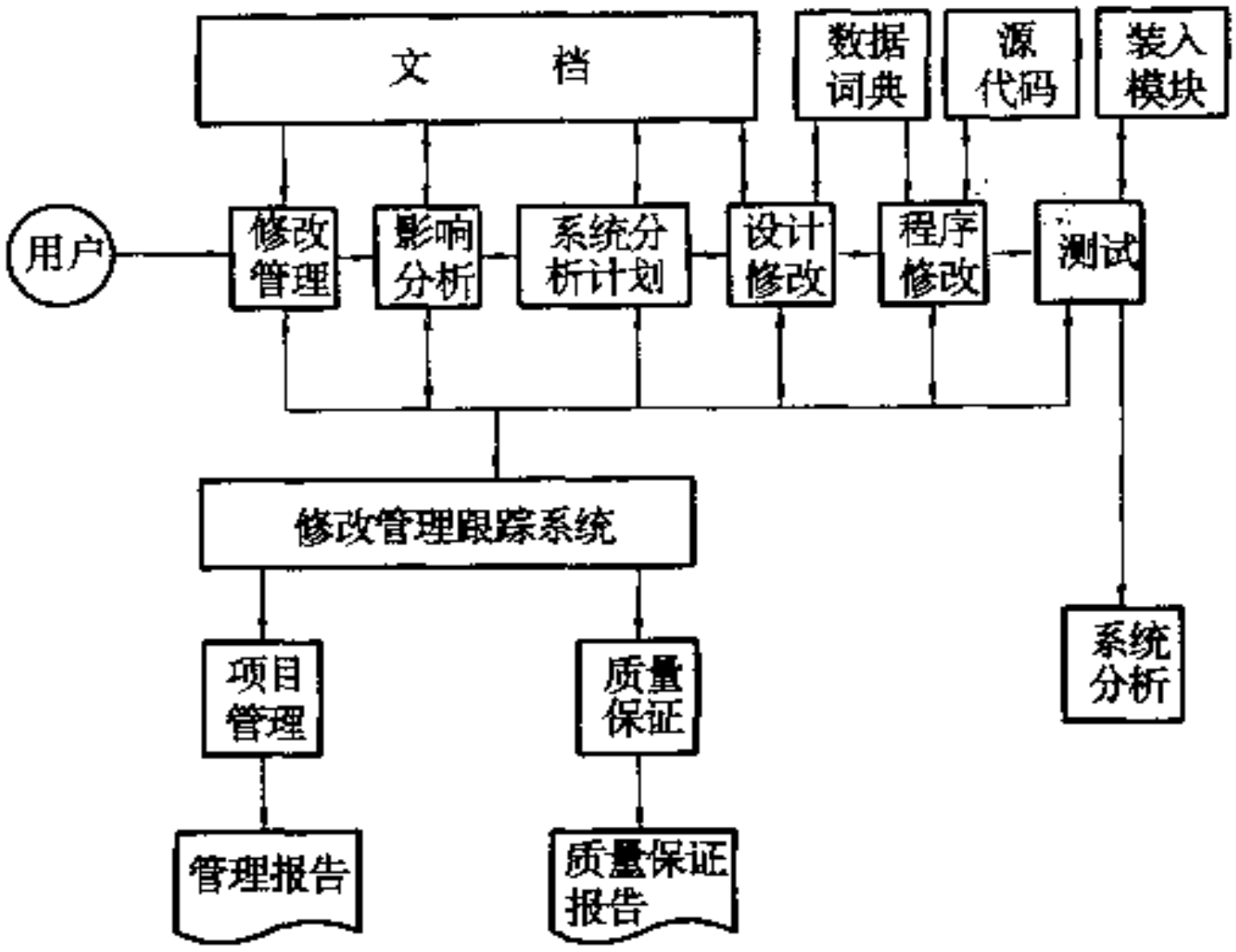


图 7.3　软件维护数据流

护人员的努力，同时，相关的管理部门、管理人员以及软件开发人员，都应加以重视，以求得妥善的解决。这些问题表现在以下几个方面：

① 许多软件的维护工作非常困难。原因在于这些软件的文档和源程序难于理解，又难于修改。从原则上讲，软件开发工作应严格按照软件工程的要求，遵循特定的软件标准或规范进行。但实际上往往由于种种原因并不能真正做到。例如，文档不全、质量差、开发过程中不注意采用结构化设计方法、忽视程序设计风格等等。总之，在开发阶段并未考虑到维护工作可能遇到的问题，也就不可能为维护工作提供任何方便。

② 软件维护的成本高得惊人。本章前一节已给出了开发与维护成本比例的变化情况，这一变化告诉人们，要维护好软件所花的费用越来越大。美国某一空军软件项目，开发出的源程序成本为

每行代码 75 美元，而它的维护工作花费为每行代码 4000 美元。另据美国国防部的 1976 年统计，60%到 70%的软件费用花在软件维护方面。

③ 维护工作面广，维护工作量大。新开发的软件不断涌现，每一个软件都需要做维护工作。维护工作量越来越大，对维护工作的需求远远超过了实际可能提供的维护能力。依靠原来的开发人员进行维护，存在着一些实际困难。

④ 由于上述的一些困难，致使维护力量薄弱，维护工作的质量直接受到影响。在维护中对所做的修改，常常考虑得不够周密，使得在修改过程中给软件带来了新的问题或引入了新的差错。或者维护中不重视文档的细节变动，使得文档不能反映维护变动后的情况，为用户带来了新的困难，并且也不利于做进一步的维护工作。

(2) 影响维护工作的因素

增加维护工作量和维护工作难度的因素包括：

- 软件的规模偏大
- 维护的软件其开发时间较早
- 软件的复杂性较大
- 应提供的用户报告数量较多
- 原软件文档的质量较差
- 软件的应用领域常变

还有一些因素对维护工作是有利的，如：

- 软件开发中采用了结构化方法
- 利用了新的软件或自动的开发工具
- 具有较好的数据管理
- 开发所用语言便于维护
- 具有一定的维护实践经验

(3) 解决维护工作困难的出路

针对上述维护工作中存在的一些矛盾，提出以下解决的对策：

① 在软件开发的开始阶段便应建立维护的观念。要使软件开发人员了解维护工作的困难，使他们具有“一定要开发出便于维护的软件”的思想，用以指导他们的分析、设计和实现的各项开发活动。在开发工作中也应在软件中采用一些具体的设施为将来的维护服务。

② 使程序结构的复杂性降低到最小。

③ 开发中坚持按结构化方法进行设计。

④ 开发中努力提高软件的可靠性，以减少改正性维护的工作量。

⑤ 开发时最好能预计到未来使用中可能的变动，使设计具有可修改、可扩充的灵活性。例如，设计中在模块划分时，把固定不变的和可能变动的部分分开。

⑥ 应注意提高文档编制的质量。

⑦ 加强维护的管理，确保维护中对变更的控制和对变更的审查。

7.3 可维护性及其度量

(1) 软件可维护性

所谓软件的可维护性(maintainability)是指，阅读软件时，易于理解的程度，在运行中发现其中的错误或缺陷时，准备加强其功能，改善其性能，而需对它做修改、变更的难易程度。软件的可维护性、可用性及可靠性构成了衡量软件质量的几个主要尺度，也是用户十分关心的几个方面。可惜的是影响软件质量的这些重要因素，目前尚不能对它们进行定量分析。然而，就它们的概念和内涵来说则是很明确的。

软件的可维护性按 B. W. Boehm 的定义，包括三方面内容，即

· 可测试(testability)

- 可理解性(understandability)
- 可修改性(modifiability)

对它们的影响因素可从图 7.4 中看出，这八个影响因素是：

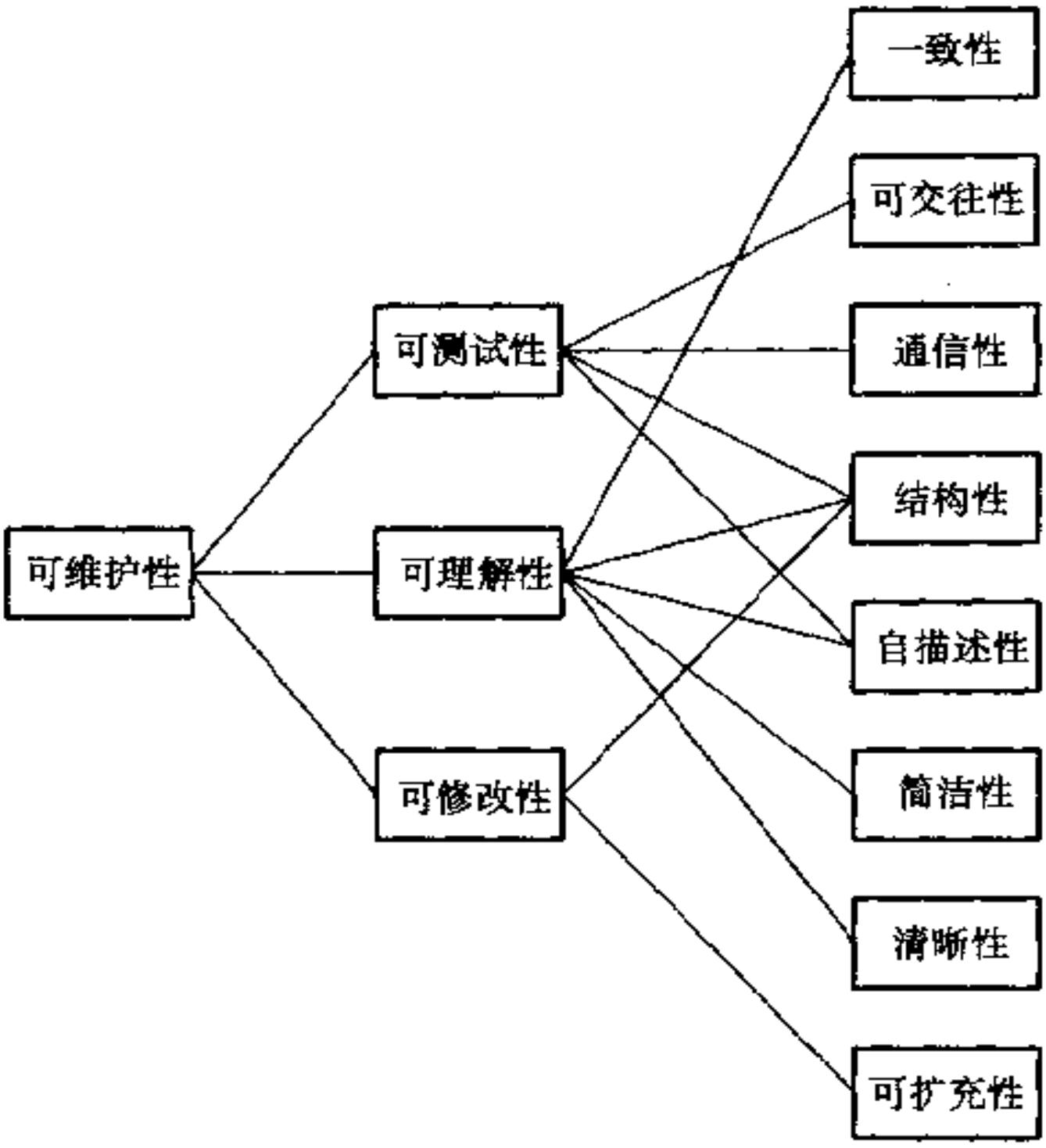


图 7.4　构成软件可维护性的因素

① 一致性(consistency)：程序及与其相关的文档所用的记法、记号以及内容是否协调一致。

② 可交往性(accessibility)：是否能较容易地选择和利用软件所具有的功能和设施。

③ 通信性(communicativeness)：对软件的输入和输出信息是

否灵活方便。

④ 结构性(structuredness)：程序结构是否符合结构化设计的要求。

⑤ 自描述性(Self-descriptiveness)：程序内的注释、符号名的命名是否能清楚地表明它的功能、结构、使用及输入输出信息等特性。

⑥ 简洁性(conciseness)：提供的信息是否都是必要的，而无多余信息。

⑦ 清晰性(legibility)：程序编写得是否具有良好的风格，便于阅读和理解。

⑧可扩充性(augmentability)：是否为进一步扩充创造了有利条件。

以上这些因素通常体现在软件产品的许多方面，为使每个因素都达到预期的高度，需要在软件开发的每个阶段采取相应的措施加以保证。即是说，这些要求实际上应渗透在各开发阶段的各个步骤当中。因此，软件产品的可维护性是产品投入运行以前各阶段面向上述各因素进行开发的最终结果。

(2) 可维护性度量

人们一直期望着对软件的可维护性作出定量的分析，但能够做到这一点并不容易。许多研究工作集中在这个方面，形成了近年来十分引人注目的学科——软件度量学(software metrics)。以下介绍两种著名的软件复杂性度量方法。由于这两个方法易于理解和使用，两者都已为软件产业部门所接受。

我们知道，程序的复杂性是程序可理解性的另一种度量方法。实际上，程序如果越复杂，人们就越难理解它。另一方面，软件维护工作势必要对被维护的软件进行某种程度的修改或更动。一个很自然的问题是，更动以后对程序复杂性带来什么影响。究竟是使程序变得更复杂了，更难理解、更难做进一步维护了，还是相反。很显

然，我们不希望经过维护变动以后的程序破坏了原来的良好结构性，如果更加复杂、更难理解、程序的质量更差了，只能说维护工作完成得不好。

① Halstead 程序工作量公式

给出源程序以后，根据程序中的某些特性，得到某些参数，按照 Halstead 公式可以求得程序工作量值。所用的程序参数有四个：

N1：程序中运算符的总数

N2：程序中运算对象的总数

n1：程序中出现的运算符种类数

n2：程序中出现的运算对象的种类数

Halstead 定义了以下几个量：

• 程序的长度

$$N=N1+N2$$

此值也可用估算值 $\hat{N}$ 代替

$$\hat{N}=n1\log_2 n1+n2\log_2 n2$$

事实上，$\hat{N}$ 和 N 是十分相近的。按下面 FORTRAN 程序的实例计算，得到 $N=50$，$\hat{N}=52.9$

• 程序量

$$V=(N1+N2)\log_2(n1+n2)$$

此值表示程序中所含信息量(如二进制位的位数)，它和程序所用的语言有关。在下面 FORTRAN 程序实例中，$V=204$，但若用汇编语言写出该程序，则 $V=328$，表示要占用更多的内存空间。

• 语言抽象级别

$$L=(2*n2)/(n1*N2)$$

• 程序工作量

$$E=V/L$$

即

$$E=(n1*N2*(N1+N2)*\log_2(n1+n2))/(2*n2)$$

以 FORTRAN、PL/1 和 APL 等几种语言编写的程序，按此公式计算，发现算出的值和实际值非常相近。例如，计算出预计要用 22.51 小时的程序工作量，实际上用了 20.15 小时。

这里给出一个用交换元素方法实现的 FORTRAN 排序程序实例：

```
      SUBROUTINE SORT(X,N)
      DIMENSION X(N)
      IF (N.LT.2) RETURN
      DO 20 I =2, N
         DO 10 J=1, I
         IF(X(I).GE.X(J)) GO TO 10
         SAVE =X(I)
         X(I)=X(J)
         X(J)=SAVE
10    CONTINUE
20    CONTINUE
      RETURN
      END
```

将 4 个程序参数值代入计算，得到的程序工作量值 E=204。

程序工作量的数值表明了开发程序所需付出的智力劳动量，也可以认为，这一数值反映了要读懂程序需要花的功夫。因而，对于软件维护来说，这个值是很有意义的。

Halstead 程序工作量计算方法的另一应用是在程序相似性检验上。如果两个程序的参数 n1，n2，N1 和 N2 值完全相同，并且程序的行数、使用的变量个数、说明了的变量个数以及控制结构的个数都完全相同，此种情况是极为罕见的。可能两者是同一程序，或者其中之一有“抄袭”的嫌疑。显然，相似性检验工具在软件保护技术中有它的重要作用。

运算符	个数	运算对象	个数
1. 可执行语句末尾	7	1. X	6
2. 数组下标	6	2. I	5
3. =	5	3. J	4
4. IF	2	4. N	2
5. DO	2	5. 2	2
6. ,	2	6. SAVE	2
7. 程序末尾	1	7. 1	1
8. .LT.	1	n2=7 N2=22	
9. .GE.	1		
10. GO TO	1		
n1=10 N1=28			

② McCabe 环路度量公式

McCabe 观察到，理解程序的困难很大程度上是由于程序控制流的复杂性。在程序流程图中，单一流线的程序结构最简单。而循环和判断所构成的环路越多，程序就越复杂。

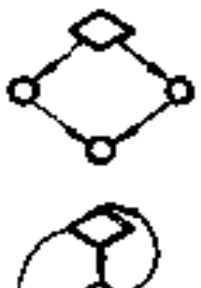
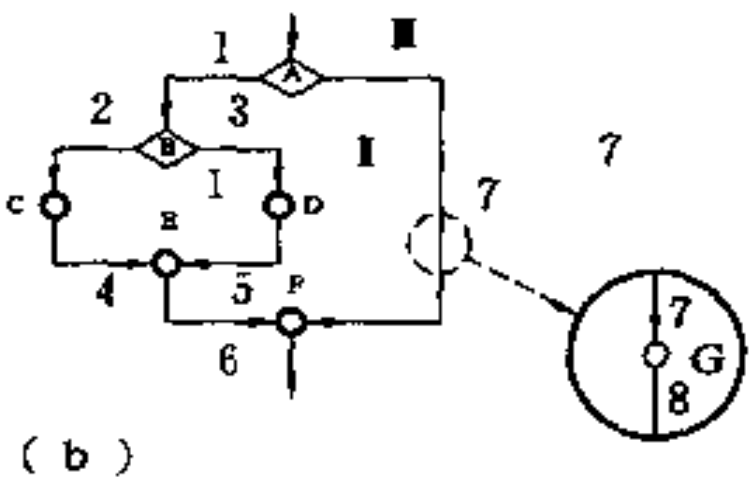
我们知道，如果把程序流程图的箭头方向去掉，只保留流线，则成为无向图。原来流程图箭头的始发点和指向点便成为无向图的节点。我们把无向图中任何两个节点之间至少存在一条通路的图称为连通图(connected graph)。McCade 提出，程序流程的连通图 G 中的环路数 V(G)可按以下公式计算：

$$V(G)=E-n+2$$

其中，E 为图 G 中的边数

n 为图 G 中节点数，这里节点指的是原流程图中的分支点、汇合点和处理框。

图 7.5 给出了几个环路计算的实例。其中(a)图包括了四种基本控制结构，(b)图是一个简单的结构化程序。注意在(b)图的 7 号边上如果增加一个节点 G，其环路数仍为 3。这是因为同时也增加了一个号码为 8 的边。

流程图		E	n	V=E−n+2
顺序型		1	2	1
选择型		4	4	2
WHILE 循环		3	3	2
UNTIL 循环		3	3	2
(a)				
		7	6	3
(b)				

V=8−7+2=3

图 7.5 程序环路计算

(a) 四种基本控制结构

(b) 简单的结构化程序

McCabe 还发现，具有单个入口和单个出口的结构化程序，其环路数 V 值恰好为程序中谓词个数加 1(在图 7.5 中即为菱形分支点的个数加 1)。并且，V 值还等于各封闭边将平面所分割的域数。例如，图 7.5 的(b)中，封闭的边将平面分割成 Ⅰ、Ⅱ、Ⅲ 三个域。

显然，任一程序如果其环路数 V 越大，则程序越复杂。它可以作为衡量程序复杂性的一个尺度。如果把具有多种选择的 CASE 型结构作为例外，不予考虑，McCabe 建议：在一个程序模块内，代表其复杂度的环路数 V 一般控制在 10 以内是适当的。V 的取值若是在 3 到 7 的范围内，被认为是良好的结构，恰当的复杂度。

无疑，McCabe 的环路计算方法在软件维护中是很有用的。

7.4 软件维护的管理

软件产品的维护工作不仅是技术性的，它还需要大量的管理工作与之配合，才能保证维护工作的质量。

原则上讲，维护工作从理解软件开始。若是对被维护的软件没有很好的理解，也就谈不到维护。这种理解包括对功能性能的分析和理解、对原设计的分析和理解以及对源程序的分析 和理解。在此基础上，如有明确的维护任务，例如，改正错误、适应性更动或是扩充性更动，都应提出维护修改建议。进一步的维护工作都应在管理部门的参与和控制下完成。图 7.6 表明了软件维护的管理流程。管理部门应对提交的修改建议进行分析和审查，并对修改带来的影响作充分的估计。对于不妥的修改予以撤销。通过审批的修改方案，经修改后应做严格的测试，以检验修改的质量。经管理部门再次审查后，方可对文档的主文本进行更新。

为确保维护中修改的正确性，消除因修改不当给用户带来的不良影响，要求对修改工作持谨慎态度。例如，维护修改建议常常提得不够具体，或是用户提出的修改要求不一定恰当。有些要求则

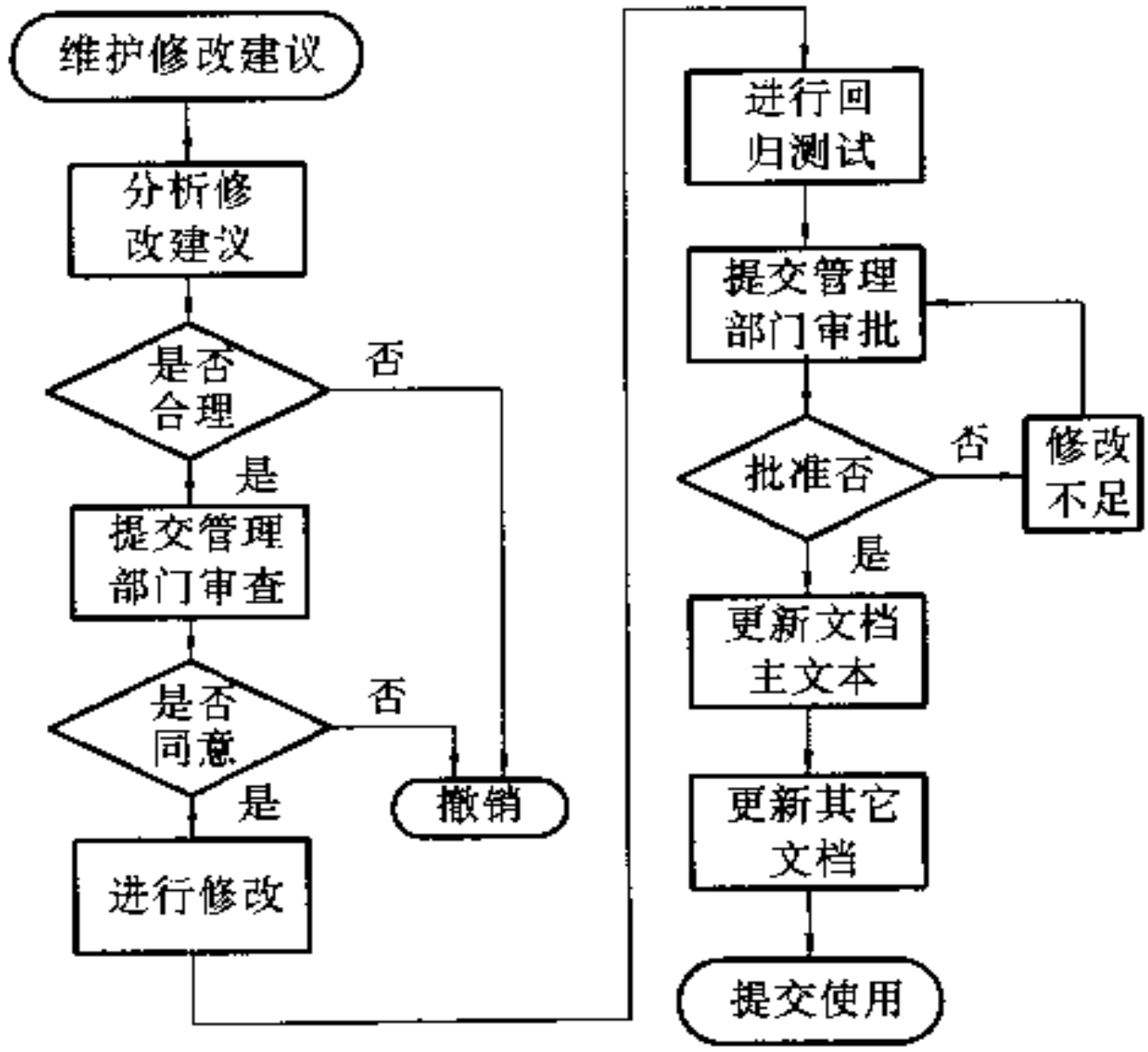


图 7.6　软件维护的管理流程

超出了软件维护工作的范围。这时分析员应和用户仔细讨论，在说明情况、弄清要求的基础上，提出修改建议。

如果仔细研究图 7.6 的维护过程，并和开发过程作一对比，我们会发现，维护工作实际上包含了需求分析、设计、编码和测试等开发软件所经历的全过程。这一点从图 7.3 也可得到进一步理解。

由谁来承担维护工作是软件维护管理的另一个问题。一般认为应该由开发软件的人员去维护。这样做的好处是他们对自己开发的软件最熟悉，维护起来更方便。如果在开发阶段就已经明确了，他们将来还有维护的责任，他们定会在开发中设法尽可能提高软件的可维护性。但这也同时会出现弊端，他们很可能不认真编写文档，以为自己去维护，文档可以马虎一点。当然，由开发人员做

维护工作势必使得他们的新项目开发工作受到影响，或者完全没有时间和精力去开发新的项目。

另一种做法是安排专职的维护人员负责维护工作。软件开发人员不管维护，他们便可集中精力做好开发工作，使新项目的开发工作不受维护工作的影响。这样做还将有利于坚持实施开发标准，有利于保证文档的编制质量。而专职的维护人员可以把被维护的软件分析得非常透彻，对其中的每一个细节都弄得非常清楚，成为这个软件的专家。不过，实际上许多人对维护工作不感兴趣，他们宁愿花力气去开发软件，认为这才是开创性的工作，维护工作没有什么水平。其实这是一种片面的认识。

还有一种较好的做法是安排软件人员实行开发任务和维护任务定期轮换。这样可以免除上述方法的某些缺点，并可使软件人员体会到开发工作和维护工作的具体技术要求，以及两者之间的相互关系。有利于提高软件人员的技术水平和软件产品的质量。

无论采用上述哪一种维护工作的组织方法，考虑如何安排维护人员的工作时，对每一个软件至少指定两个人负责它的维护工作是很必要的。这样做的好处是，避免了软件维护工作对某个人的过分依赖，防止由于工作调动等原因，维护工作受到影响而落空。

有些软件开发部门对已交付用户的软件提供有限的“售后”服务。例如，有的公司对于售出的系统软件区分以下几种情况，采取不同的对策：

① 已交付使用的软件并非用户方面的原因，造成运行失效，这种情况一经确认，开发部门有责任在最短时间内限期排除故障，比如几天以内。

② 交付使用的软件出现故障，但仍可运行，并未对用户构成严重干扰。开发部门答应在两至三周内予以排除。

③ 软件投入使用后，用户提出了增强功能、改善性能方面的要求，开发部门通常难于立即接受。但可能把这些要求置于下一版

本修订工作开始时予以考虑。

④ 用户提出的新要求并非十分必要，但可能给用户带来某些方便。开发部门将酌情予以考虑。

软件工程项目的实践表明，维护工作远不如开发工作对软件人员有吸引力。人们常常宁愿参加“开拓性”的开发工作，而不愿做修修补补的维护工作，事实上，做好维护工作是困难的。但当我们仔细分析维护工作的困难时，就会发现，这些困难往往是由于开发工作的缺陷所造成的。比如，开发中不能严格地遵循标准或规范的有关规定，不重视文档编制工作，程序的可读性差等等。总之，未能按照软件工程的原则去做，把一些问题隐藏了起来。在测试和阶段评审时也只能解决其中的一部分问题，软件运行中暴露出另一些问题，但解决起来并不像拆换一个零件那么简单。软件的维护不能完全指望原来的开发人员总是“守护”着它的运行，必须认真组织专职的维护人员队伍。在软件运行中尚未暴露出问题时，维护人员应着重于熟悉掌握软件的有关文档，一旦维护任务提出来，他们就应高质量地完成维护工作。

第八章　软件管理

近年来，工程技术的发展提出了许多与管理有关的课题，形成了新的技术领域。我们如果把一般意义的工程技术，如计算机技术、半导体技术、生物工程技术等称为固有技术，那么，工业管理、工程经济、运筹学等统称为管理技术。然而，对管理技术的认识还是逐步加深的。传统的观念常常使人们只看到具体、有形的固有技术，而忽视抽象、无形的管理技术。事实上，只靠固有技术，工程或科研项目的效率、质量、成本和进度等问题很难得到较好的解决。另一方面，管理技术即使有条件从国外引进，若想让它发挥效益，还必须结合我们自己的工作条件、人员及社会环境等多种因素。简单地搬用国外的管理技术往往是无法奏效的。此外，管理技术的基础是实践，为取得管理技术的成果必须花功夫，付学费，反复实践。十分明显，管理能够带来效率，能够赢得时间，最终将在技术前进的道路上取得领先地位。在知识爆炸、高科技迅速发展的今天，我们必须以战略的眼光来看待技术管理问题。

当前软件管理，尤其应当特别强调。原因在于至今人们对软件管理问题还没有足够的了解。近几年对软件的作用和地位开始得到了承认，算是从原来只重视计算机硬件的状况前进了一步。然而，如何看待软件的技术工作和管理工作仍然需要有一个正确的认识。鉴于软件产品和软件项目的特点，软件管理的作用显得格外重要，在一定意义上讲，它是软件产业发展的关键。软件项目的规模越大，所需要的管理支持工作量越大。统计资料表明，在软件项目的规模达到一定大小时，所需要的软件管理工作量达到总工作量的一半(参看图 8.1)。

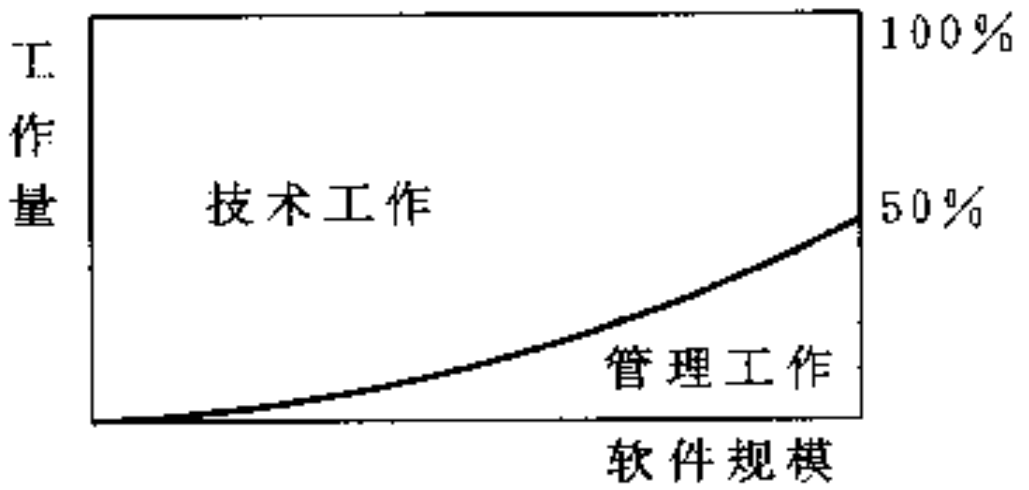


图 8.1　软件管理工作量与软件项目规模的关系

同时，我们注意到，软件管理问题的解决具有一定的难度，它可能涉及到系统工程学、统计学、心理学、社会学、经济学乃至法律等。需要用到多方面的综合知识，特别是要涉及到社会的因素、精神的因素、人的因素，也就要比技术问题复杂得多。

在讨论软件管理之前，有必要对软件管理的对象，对软件项目和软件工程本身的特点作些分析。既是为展开管理问题的讨论作好准备，也是把前述软件生存期各阶段工作做一概括性的小结。然后就软件管理的几个重要问题，包括如何组织好软件工程项目，软件配置管理以及软件成本估算等问题给出简要的介绍。

8.1　软件项目的特点和软件管理的职能

(1) 软件项目的规模

软件项目的规模决定了采用怎样的管理水平、开发工具和开发方法。正如同建筑师只有明确了建筑任务是盖一间小屋还是盖一座几十层大厦以后，才能考虑怎样去组织设计和施工工作一样。

著名软件专家 Yourdon 对软件项目的规模给出了以下的分类方法。这种分类 集中体现在表 8.1 中。

微型：只是一个人，甚至是半时，在几天之内完成的软件项目。写出的程序不到 500 个语句行，仅供个人专用。也许用过以后就丢掉了，通常这种小题目无需做严格的分析，也不必要有一套完整的

设计、测试资料。不过这并不是说可以随便地不讲任何方法地做。事实说明，即使这样小的题目，如果经过一定的分析、系统设计、结构化编码以及有步骤地测试，肯定也是非常有益的。

表 8.1　软件项目规模的分类

分　类	参与人员数	研制期限	产品规模（源程序行数）
微　型	1	1—4 周	500
小　型	1	1—6 月	1K—2K(1K＝10^3)
中　型	2—5	1—2 年	5K—50K
大　型	5—20	2—3 年	50K—100K
甚大型	100—1,000	4—5 年	1M(1M＝10^6)
极大型	2,000—5,000	5—10 年	1M—10M

小型：一个程序人员半年之内完成的两千行以内的程序。例如，数值计算问题或是数据处理问题。学生做的毕业设计正是这种规模的课题。这种程序通常没有和其它程序的接口。但已经需要按一定的标准化技术、正规的资料书写以及定期的系统审查了。只是没有大题目那么严格。

中型：5 人以内在一年多时间里完成五千到五万行程序。这种课题开始出现了软件人员之间，软件人员和用户之间的联系、协调的配合关系问题。因而，计划、资料书写以及技术审查就要比较严格地进行。这种类型的软件课题是比较普遍的。许多应用程序和系统程序就是这样的规模。在研制中使用系统的软件工程方法是完全必要的。并且对于提高软件产品质量和程序人员的工作效率，以及更好地满足用户的要求起着重要的作用。

大型：5 至 20 人在两年多时间里完成五万至十万行程序。比如编译程序、小型分时系统、应用软件包、实时控制系统等很可能

是这种大型软件。参加工作的软件人员需要按二级管理，比如划分成若干小组。每组5人以下为好。在任务完成过程中，人员调整往往不可避免。因此会出现对新手的培训和逐步熟悉工作的问题。对于这样规模的项目，采用统一的标准，实行严格的审查是绝对必要的。由于项目的规模庞大以及问题的复杂性，往往会在研制过程中出现一些事先难于作出估计的不测事件。

甚大型：一百人至一千人参加用4到5年时间完成具有万行程序的项目。这种甚大型项目可能会划分成若干个子项目，每一子项目都是一个大型软件。子项目之间具有复杂的接口，例如，实时处理系统、远程通讯系统、多任务系统、大型操作系统、大型数据库系统、军事指挥系统通常有这样的规模。很显然，这类问题没有软件工程方法的支持，它的开发工作是不可想像的。

极大型：两千到五千人参加，十年内完成千万行以内的程序。这类项目是很少见的，往往是军事指挥、弹道导弹防御系统。

从上述分类的情况可看出，规模大、时间长、很多人参加 的项目，其研制工作必定要有软件工程的知识作指导。而规模小、时间短、参加人员少的项目也得有软件工程概念，遵循一定的开发规则。其基本原则是一个，只是对软件工程技术依赖的程度不同罢了。

(2) 究竟什么是软件工程

Boehm 曾经为软件工程下了定义：运用现代科学技术知识来设计并构造计算机程序及为开发、运行和维护这些程序所必需的相关文件资料。这里对“设计”一词应有广义的理解，它应包括软件的要求分析以及对软件进行修改时所进行的再设计活动。1983 年 IEEE 给出的定义为：软件工程是开发、运行、维护和修复软件的系统方法，其中“软件”定义为，计算机程序、方法、规则、相关的文档资料以及在计算机上运行时所必需的数据。Fairley 认为，软件工程是为在成本限额以内，按时完成开发和修改软件产品所需的系

统生产和维护的技术和管理的学科。

十分明显，软件工程的主要目标是改进软件产品的质量，提高软件研制的生产效率。它和管理学科的关系非常密切。

(3) 软件项目的特点

软件产品和其它任何产业的产品不同，它是无形的，既没有质量，没有体积，也没有颜色，没有气味——完全没有物理性质，对于这样看不见、摸不着的产品让人难以理解、难于驾驭。但它确是把算法、思想、概念、组织、流程、效率、优化等融合成一体了。

要开发这样的产品，在许多情况下，用户一开始给不出明确的想法，提不出明确的要求。他叙述不清，究竟他需要的是什么。

在开发的过程中，程序和与其相关的文档资料常常需要修改。在修改的过程中又可能带来新的问题，并且这些问题很可能在过了相当时间以后才发现。

文档资料工作的工作量在整个研制工作中占有很大比重，是十分重要的工作，但从实践中看出，人们对它不感兴趣。认为是不得不做的苦差事，不愿认真去做。因而直接影响了软件的质量。

软件开发工作技术性很强，要求参加工作的人员具有一定的业务水平和实际工作的经验。但事实上，人员的流动对工作的影响很大。离去的人员不仅带走了重要的信息，还带走了工作经验。

(4) 软件管理的特殊困难

① 智力密集，可见性差

软件工程活动充满了大量的高强度脑力劳动。软件开发的成果是不可见的逻辑实体，软件产品的质量难于用简单的尺度加以度量。对于不深入掌握软件知识或缺乏软件开发实践经验的人员，不可能做好管理工作。软件开发任务完成得好也看不见，完成得不好有时也能制造假象，欺骗外行的领导。

② 单件生产

在特定机型上，利用特定的硬件配置，由特定的系统软件或

支撑软件的支持，形成了特定的开发环境。再加上软件项目的特定目标，采用特定的开发方法、工具和语言，使得软件产品具有独一无二的特色。几乎找不到与之完全相同的软件产品。这种建立在内容、形式各异基础上的研制或生产方式与其它领域中大规模现代化生产有着很大差别，也自然会给管理工作造成许多实际困难。

③ 劳动密集、自动化程度低

软件工程项目经历的各个阶段都渗透了大量的手工劳动，这些劳动又是十分细致、复杂和容易出错的。尽管近年来开展了软件工具的研究，但总体来说，仍处于初期阶段。个别工具在某些阶段上能减轻人们的手工劳动，但远未达到自动化阶段。软件产业所处的这一状态加之软件本身的复杂性使得软件的开发与维护难于避免多种错误，软件的正确性难于保证，软件产品质量的提高自然受到很大影响。

④ 使用方法繁琐，维护困难

用户使用软件需要掌握计算机的基本知识，或者接受专门培训。否则面对多种使用手册、说明和繁琐的操作步骤，学会使用要花很大力气。另一方面，遇到了维护问题，如果没有配备专职维护人员，又得不到开发部门及时的售后服务，更是无可奈何。

⑤ 软件工作渗透了人的因素

为高质量完成软件工程项目，充分发掘软件人员的智力才能和创造精神，不仅要求软件人员具有一定的技术水平和工作经验，而且还要求他具有良好的心理素质。软件人员的情绪和他的工作环境，对他的工作肯定有着很大影响。与其它行业比较，它的这一特点十分突出，必须给予足够的重视。

(5) 造成软件项目失误的原因

在总结和分析足够数量失误的软件项目以后，看出其原因大多与管理工作有关。

在软件项目开发的初期往往会出现：

- 可供利用的资料太少
- 项目负责人的责任不清
- 项目的定义模糊
- 没有计划或是计划过分粗糙
- 资源要求未按时作出安排而落空
- 没有明确规定子项目完成的标准
- 缺乏使用工具的知识
- 项目已有更动，但预算未随之改变

在软件项目开发中可能会发生：

- 项目审查只注意琐事而走过场
- 人员变动造成对工作的干扰
- 项目进行情况未能定期汇报
- 对阶段评审和评审中发现的问题如何处置未作出明确规定
- 资源要求并不像原来预计的那样大
- 未能做到严格遵循要求说明书
- 项目经理人员不足

项目进行到最后阶段可能会发生：

- 未做质量评价
- 取得的知识和经验很少交流
- 未对人员工作情况作出评定
- 未作严格的移交
- 扩充性建议未写入文档资料

总之，问题涉及到软件项目研制中的计划制定、进度估计、资源使用、人员配备、组织机构和管理方法等软件管理的许多侧面。

(6) 软件管理的主要职能

软件管理的主要职能包括：

① 制定计划：规定待完成的任务、要求、资源、人力和进度等。

② 建立组织：为实施计划，保证任务的完成，需要建立分工

明确的责任制机构。

③ 配备人员：任用各种层次的技术人员和管理人员。

④ 指导：鼓励和动员软件人员完成所分配的任务。

⑤ 检验：对照计划或标准监督和检查实施的情况。

以下将针对软件管理的主要问题展开讨论。

8.2 制定计划

软件开发项目的计划涉及到实施项目的各个环节，带有全局的性质。计划的合理性和准确性往往关系着项目的成败。据美国联邦政府调查，因软件计划不当而造成项目失败占失败总数的一半以上。

计划应力求完备。要考虑到一些未知因素和不确定因素，考虑到可能的修改。

计划应力求准确。尽可能提高所依据数据的可靠程度。

(1) 制定计划的目标和风险驾驭

为了使软件开发项目取得成功，在开工以前，作好计划工作的必要性显而易见，无需赘述。制定计划的目标就是要回答：这个软件开发项目的工作范围是什么，需要哪些资源，应花费多少工作量，要用的成本有多少，以及进度安排怎 样等一系列问题。这步工作应当以系统计划为基础，以系统说明书(System Specification)为依据。尽管这样，在开发工作尚未开始以前，准确回答这些问题，显然是非常困难的。因为需 求分析还没有进行，就连一些最必要的信息也提不出来，采用估计的办法便成为不可避免的了。既然是凭着已往的开发经验做出估计，就很难达到准确，同时，从估计出发，开展的项目必然带有一定的风险。显然，估计的准确性差，风险也就愈大。进一步我们可以认为，研制的软件项目愈复杂，规模越大，结构化程度愈低，资源、成本、进度等因素的不确定性越大，承担这一项目所冒的风险也越大。组织软件开发项目

必须事先认清可能构成风险的因素，并研究战胜风险的对策。只有这样才能避免出现灾难性后果，取得项目的预期成果。当前，软件风险驾驭(Software Risk Management)如同软件质量保证和软件配置管理一样，正在形成独立的学科。它所涉及的知识包括风险估计和风险控制两个方面。

风险估计应分析哪些因素可能构成风险，其中哪些是关键因素，怎样才能避开或消除这些风险因素。风险控制则研究制定风险驾驭计划，消除风险的可能程度以及如何调整原来的项目计划等。图 8.2 表明了软件风险驾驭所涉及到的问题。

(2) 软件计划的类型

针对不同的工作目标，软件计划可能有以下多种类型：

① 项目实施计划或软件开发计划

这是软件开发的综合性计划，通常应包括任务、进度、人力、环境、资源和组织等多个方面

② 质量保证计划：把软件开发的质量要求具体规定为每个开发阶段可以检查的质量保证活动。

③ 软件测试计划：规定测试活动的任务、测试方法、进度、资源和人员职责等。

④ 文档编制计划：规定所开发项目应编制的文档种类、内容、进度和人员职责等。

⑤ 用户培训计划：规定对用户进行培训的目标、要求、进度和人员职责等。

⑥ 综合支持计划：规定项目开发过程中所需要的支持，以及如何获取和利用这些支持。

⑦ 软件分发计划：开发项目完成后，如何提供用户。

(3) 项目实施计划中任务的划分

软件项目的实施，如何进行工作的划分是实施计划首先应解决的问题。常用的计划结构有：

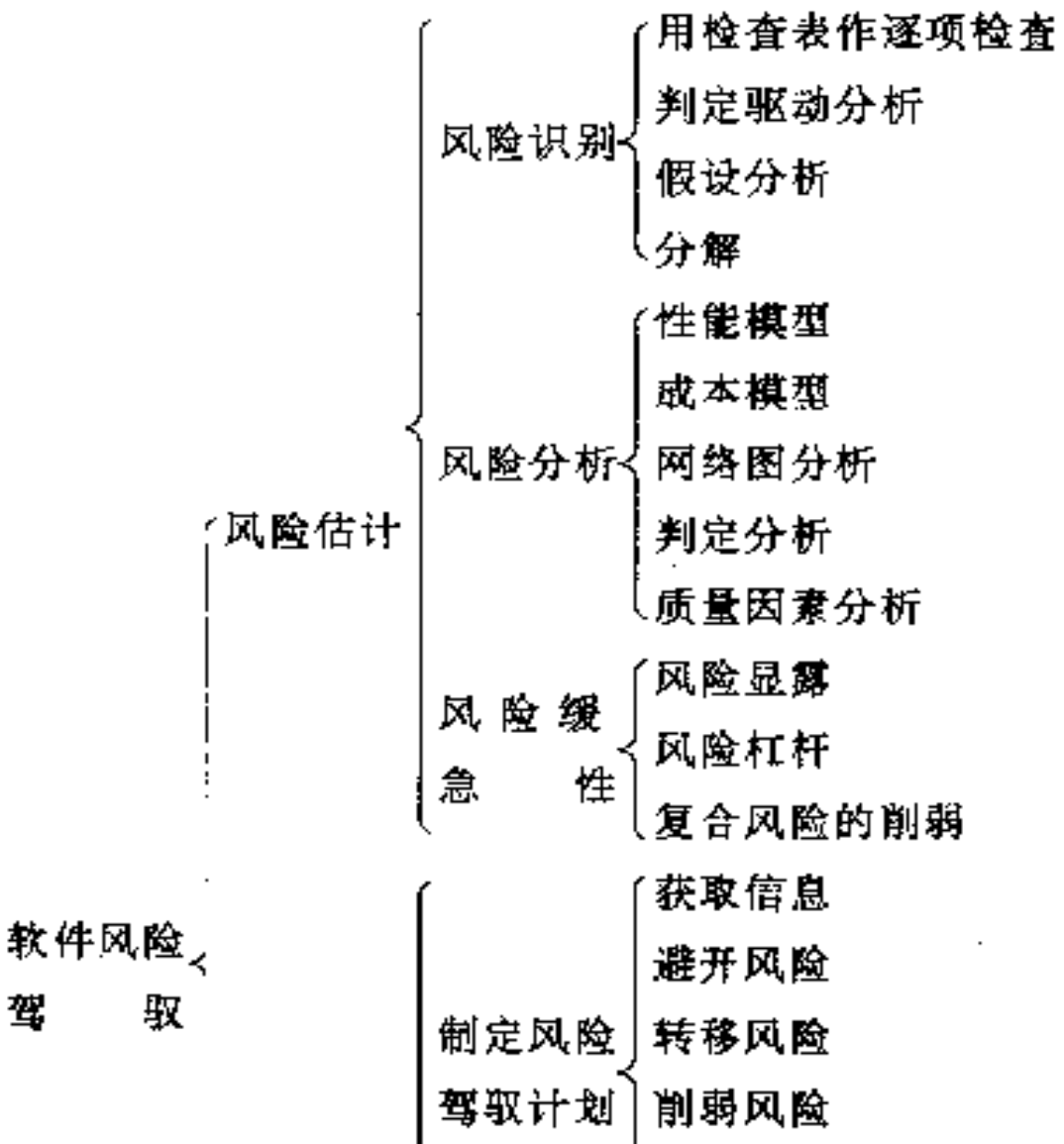


图 8.2　风险驾驭

① 按阶段进行项目的计划工作(Phased Project Planning)

按软件生存期把全部工作划分成若干阶段(究竟分成几个阶段，由管理部门具体确定)对每个阶段的工作做出计划。再把每个

阶段的工作进一步分解成若干个任务，做出任务计划。还要把任务细分为若干步骤，做出步骤计划。这样三个层次的计划便成为整个项目计划的依据。显然，过细地做好分层计划必定能提高整个计划的精确度，减 少或及早地发现问题。

② 任务分解构造(Work Breakdown Structure)

按项目本身的实际情况进行结构化的分解，自顶向下地形成树状结构。进一步把工作内容所需的工作量和预计完成的期限也规定下来。这样可以把划分后的工作落实到人，从而做到责任明确，便于监督检查(参看图 8.3)。

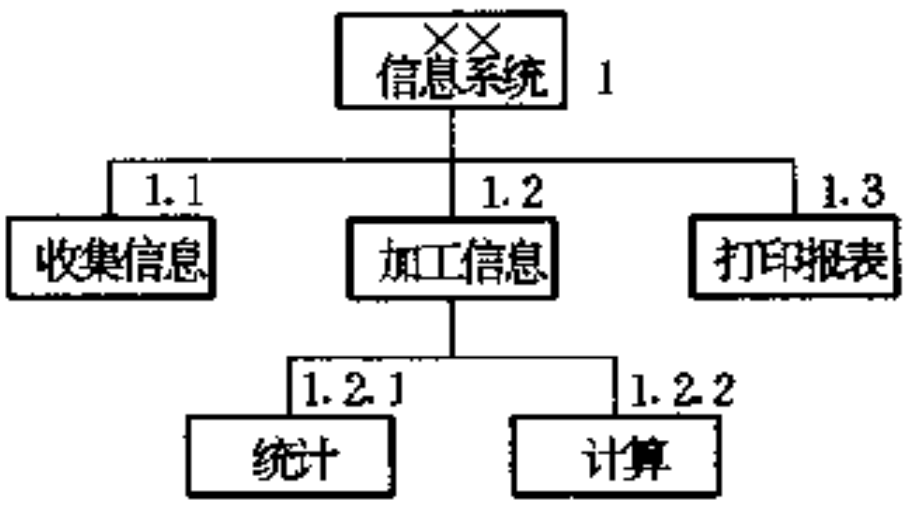


图 8.3　某信息系统任务分解

③ 任务责任矩阵(Task Responsibility Matrix)

在任务分解的基础上，把工作分配给相关的人员。这里用一个矩阵形表格表示任务的分工和责任。我们把图 8.3 已分解的任务分配给五位软件开发人员，表 8.2 表明了利用任务责任矩阵表达的分工情况。从表中可看出，工作的责任是非常明确的，并且也反映了任务的层次关系。

(4) 进度计划的控制

软件开发的组织工作是复杂的。工作的进度计划以及工作的实际进展情况，对于较大的项目来说，难以用语句叙述清楚。特别

表 8.2　责　任　矩　阵

编　号	工作划分	负责人 张××　审批	系统工程师王×× 审查	系统工程师李×× 设计	程序员 赵×× 实现	程序员 陈×× 实现
1	××信息系统	审批	审查			
1.1	收集信息		审查	设计	实现	
1.2	加工信息			审查		
1.2.1	统计		设计			实现
1.2.2	计算		设计			实现
1.3	打印报表		审查	设计	实现	

是表现各项子任务之间进度的相互依赖关系，需要采用图示的办法。以下介绍几种有效的图示方法。

① 甘特图(Gantt Chart)

甘特图以水平线段表示子任务的工作阶段，线段的起点和终点分别对应着子任务的开工时间和完成时间，线段的长度表示完成任务所需的时间。图 8.4 给出了具有五个子任务的甘特图(子任务名分别为 A、B、C、D 和 E)。如果这五个线段分别代表完成子任务的计划时间，那么在横坐标上附加一个可向右移动的纵线。它可随着时间的进展，指明扫过的已完成的子任务和尚未扫过的有待完成的子任务。我们从甘特图上可以很清楚地看出各子任务在时间上的对比关系。然而，甘特图却难以表达多个子任务之间复杂的逻辑关系。

② 时标网状图(Time Scalar Network)

为克服甘特图的缺点，我们用具有时标的网状图表示各子任务的进度情况(请参看图 8.5)。从图中可以看出各子任务间在进度上的依赖关系。例如，从甘特图中我们并不知道子任务 A 和子

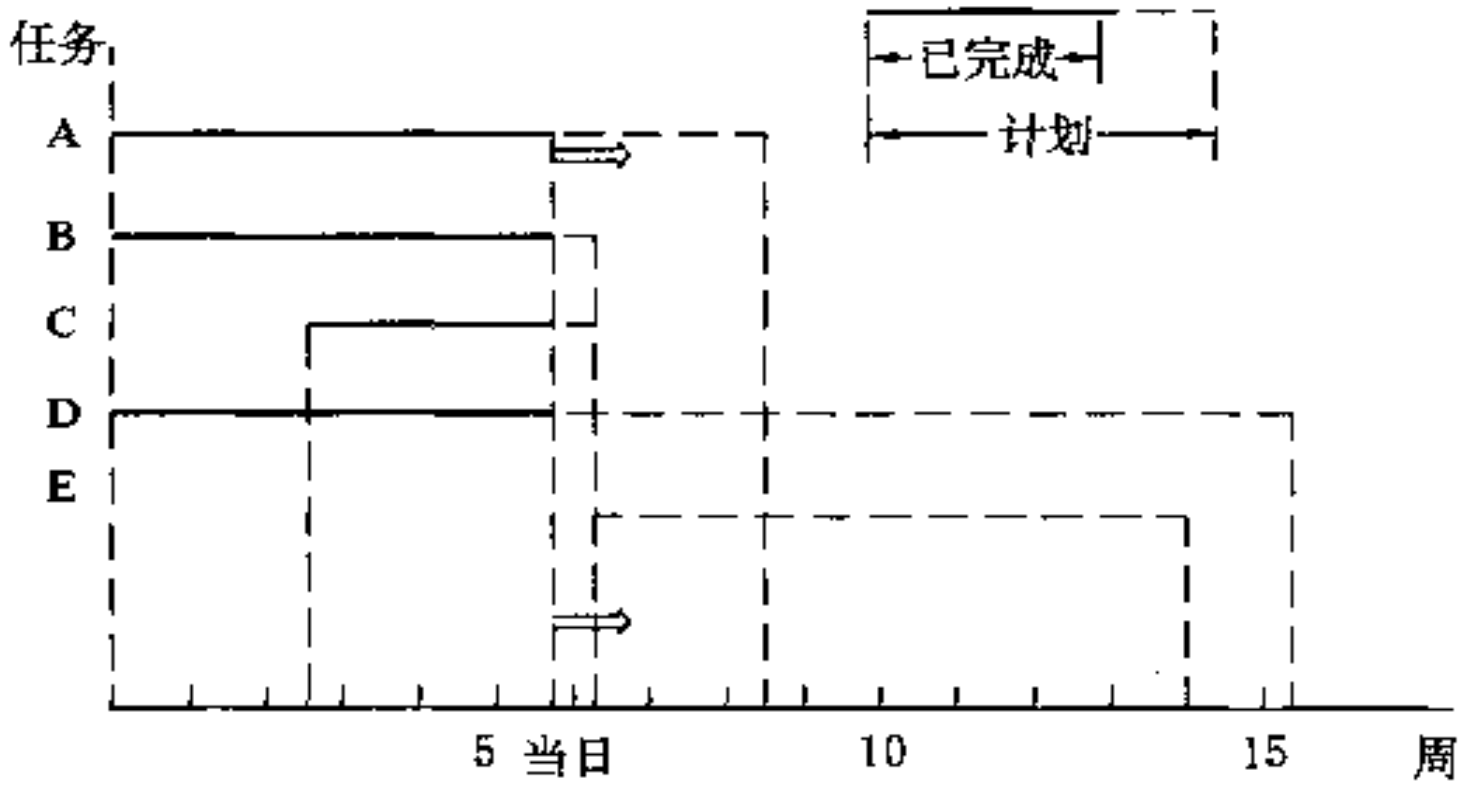


图 8.4　甘特图

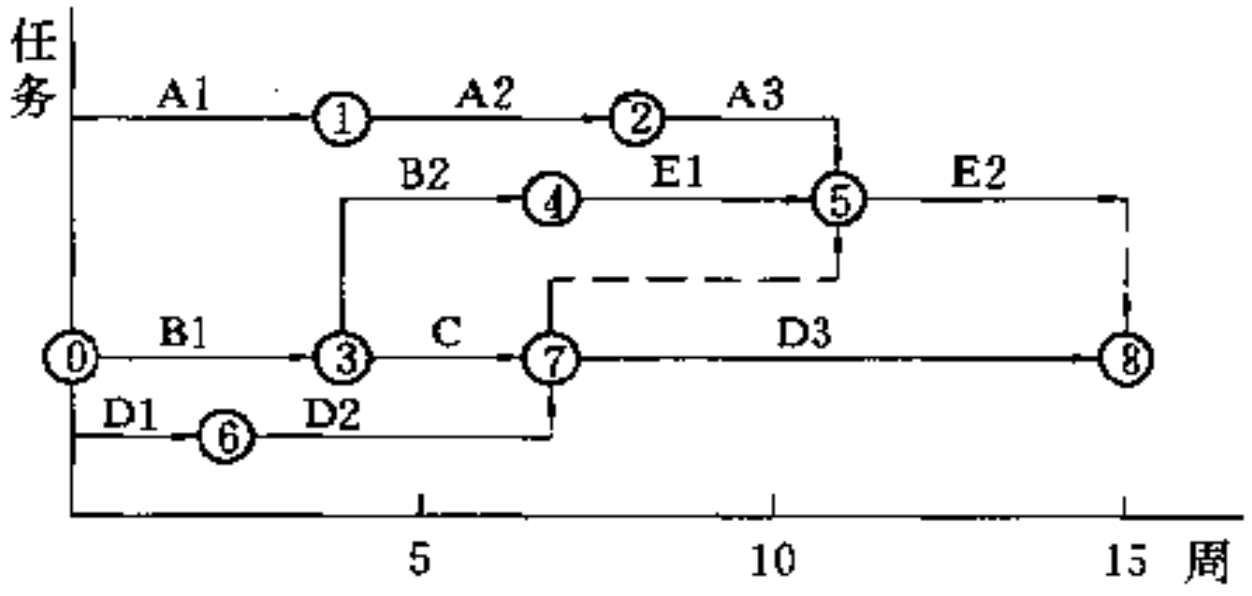


图 8.5　时标网状图

任务 E 之间有什么关系。但从时标网状图中可看出，子任务 A 分为三段，子任务 E 分为两段。E_2 的开始取决于 A_1 的完成。

③ 进度计划评审方法(Program Evaluation and Review Technique)

进度计划评审方法也称网络图方法，简称 PERT 图，是国外 60 年代初发展起来的一种先进管理技术。在国民经济各部门，如科研项目以及建筑、机械和国防工程项目中有着广泛的应用，成为

计划管理现代化和辅助决策的重要手段。让我们先来看一个软件开发的实例。假定某一开发项目，进入编码阶段以后，考虑如何安排三个模块：A、B 和 C 的开发工作。若 A 为一公用模块，模块 B 和 C 的测试有赖于 A 的调试通过。模块 C 是利用现成的已开发模块，对它需要看懂后，加以局部的修改。直到 B 和 C 做集成测试为止。这些工作步骤按图 8.6 来安排。在此网状图中，箭号表示事件，也即要完成的子任务，箭号旁均给出子任务的名称，如“A 编码”表示模块 A 的编码工作。箭号旁的数字则表明完成该项子任务的时间。图中的圆圈节点是事件的起点和终点，本图 0 号节点和 8 号节点分别表示整个网状图的起点和终点。图中足够清楚地表明了各项子任务的计划时间，以及各项子任务之间的依赖关系。

把前面甘特图和时标网状图的例子画成网络 PERT 图，如图 8.7 所示。让我们对它做进一步分析。

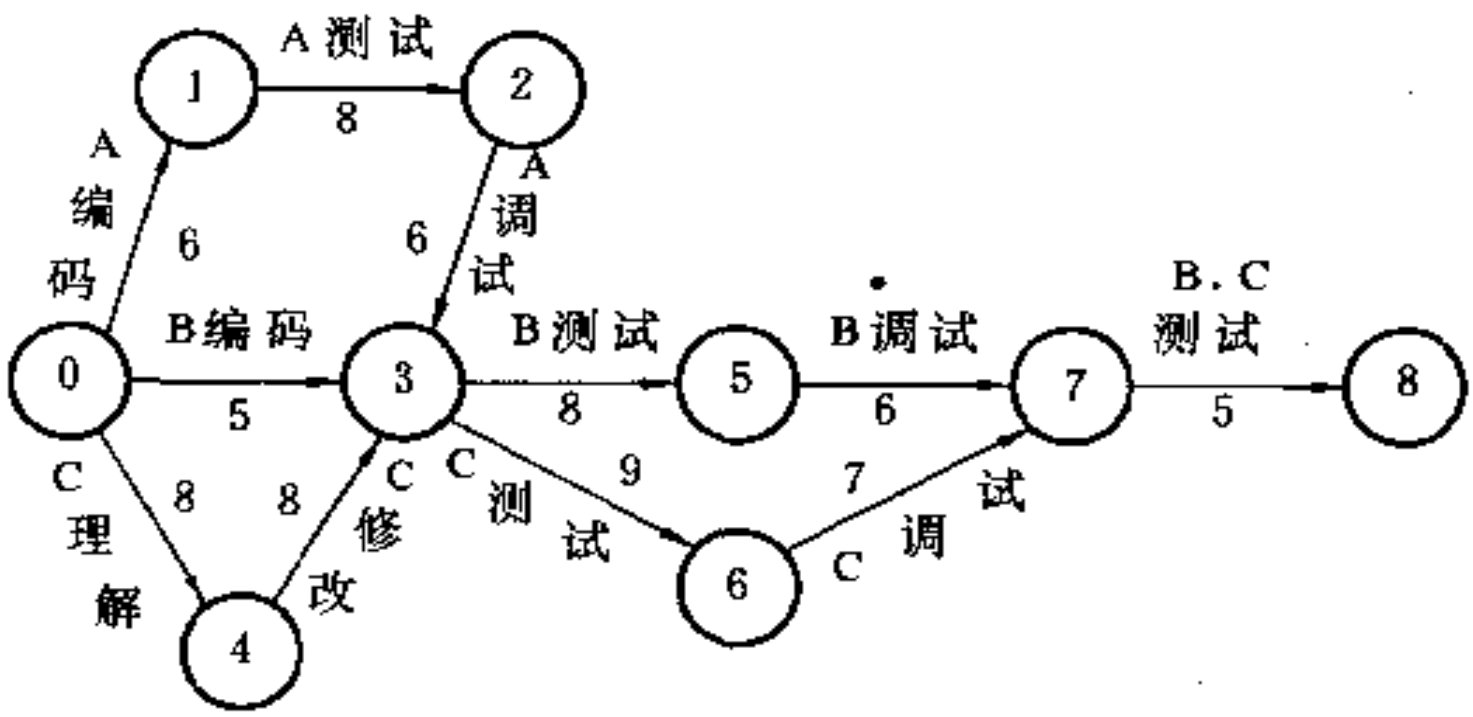


图 8.6　开发模块 A、B、C 的网状图

以节点 5 为例，从起始节点到达节点 5 有两条路径：0—1—2—5，所用时间为 9 周；0—3—4—5，所用时间为 11 周。由于子任务 E_2 要求 A_3 和 E_1 都完成以后才开始，即使由前一路径已先期到达

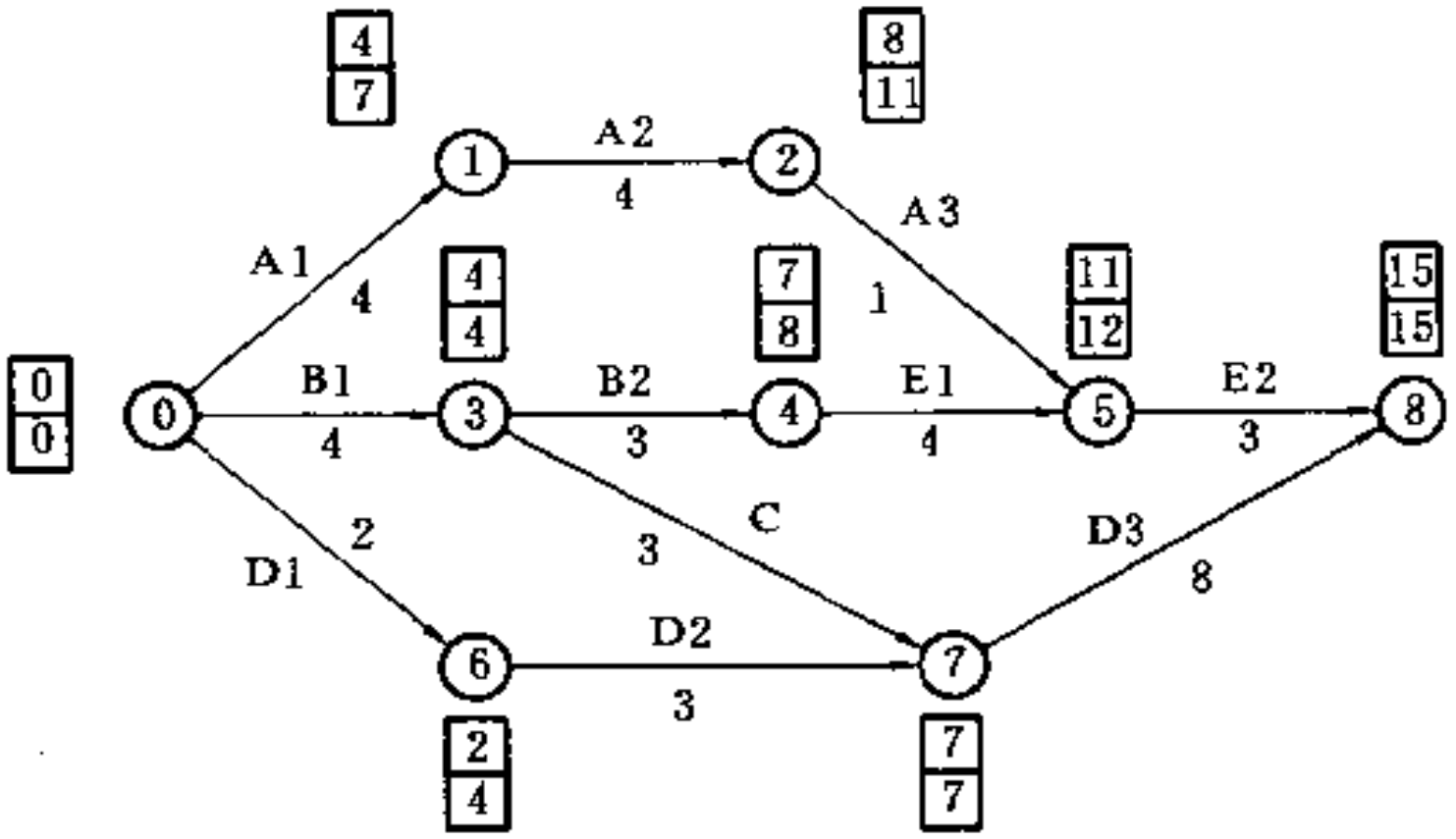


图 8.7　PERT 图

5 号节点，E_2 也不能开始，必须再等 2 周，E_1 到达后，E_2 才能开始。因此，5 号节点附近的上方框内记为 11(而不是 9)，这一数字表明该节点的最早启动时间。其它有多个箭头指向的节点均有类似情况，如节点 7 和 8。另一方面，从终点逆向推进，在不影响任务进度的条件下，可得到各节点的最迟启动时间。以节点 3 为例：沿路径 8—5—4—3 倒推至节点 3 应为 5 周启动(15—3—4—3=5)；但沿路径 8—7—3 则应 4 周启动(15—8—3=4)。因此，节点 3 最迟起动时间为 4 周，在该节点附近的下方框内记为 4(而不是 5)。依此方法给每个节点的上下方框内均填入时间。我们特别注意到：0、4、7、8 各节点的上下框内数字相同，这表明在这些节点上没有停留的机动时间，这些节点构成的路径所用时间是任务完成的最短时间，称此路径为关键路径。其它节点上两个数字不等，其差值为在此节点的机动时间。

在组织较为复杂的任务时，或是需要对特定的子任务进一步作更为详细的计划时，我们可以使用分层的 PERT 图。图 8.8 表

示，在父图 No. 0 上的子任务 P 和 Q 均已分解出对应的两个子图：No. 1 和 No. 2。

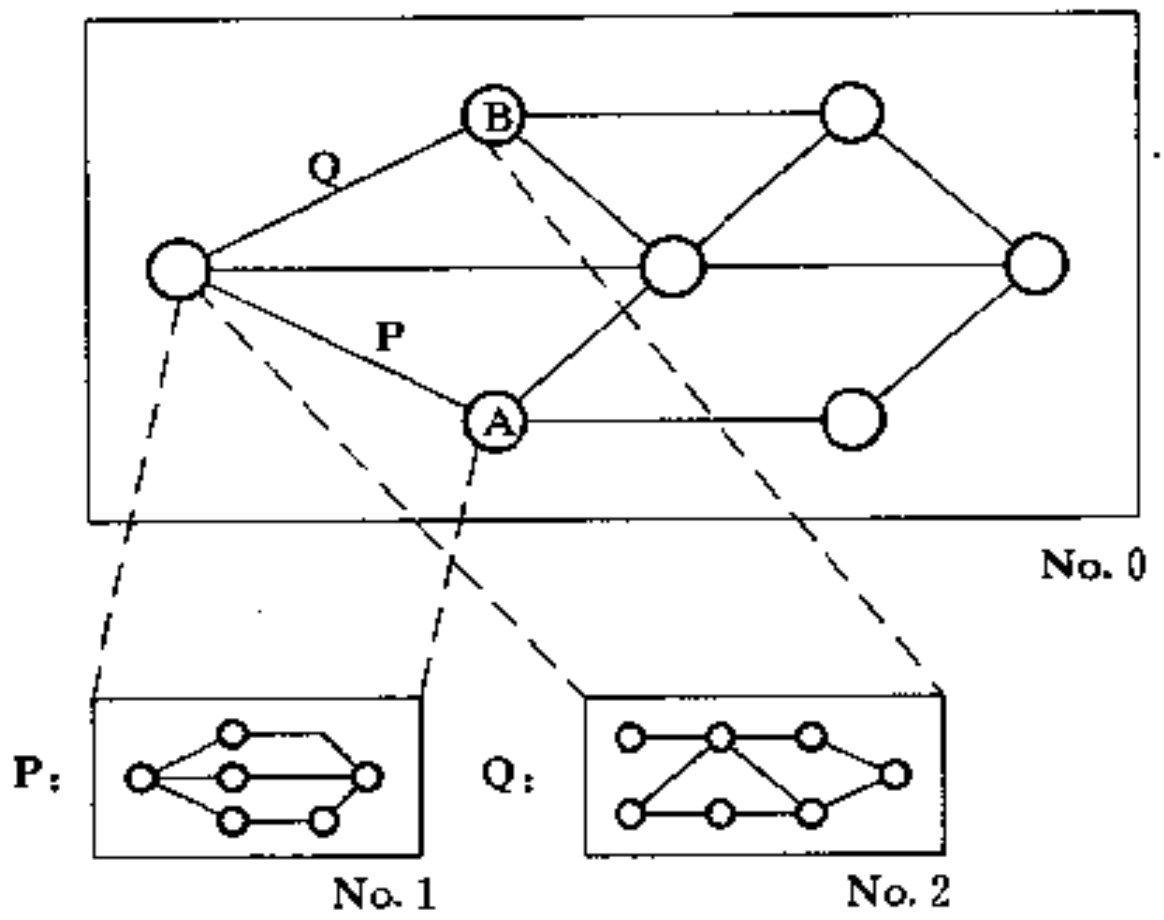


图 8.8　分层 PERT 图

PERT 图不仅可表达子任务的计划安排，还可在任务计划执行过程中估计任务完成的情况，分析某些子任务完成情况对全局的影响，找出影响全局的区域和关键子任务，以便及早采取措施，确保整个任务的按时完成。

为有效地使用 PERT 图来控制进度计划的实施，可以把这个网络图存入计算机，并配以相应的软件支持，使它成为强有力的进度计划工具。对于较大规模或较为复杂的项目，PERT 图能起到一定的指导作用。

最后，在软件工程项目中，必须处理好进度与质量这一对矛盾。我们常常有这样的经验，当任务未能执行计划而延误时，只好设法加快进度，赶上去。但事实告诉我们，在进度压力下赶任务，其成果常常是以牺牲产品的质量为代价的。但也应注意到，产品的质量和生产率有着密切的关系。例如，日本的许多产品能做

到质量和生产率的一致。日本人有个说法:在价格和质量上折衷是不可能的，但高质量给生产者带来了成本的下降这一事实是可以理解的。在日本的一些公司中，例如，日立、富士通公司中任务组(project team)对未圆满完成的产品的交付具有否决权，尽管顾客声称"我愿降低标准验收"。

8.3 建立组织

参加软件开发的人员如何组织起来，使他们发挥最大的工作效率，对成功地完成软件项目极为重要。开发组织采取什么形式要针对开发项目的特点来决定，同时也和参加工作的人员素质有关。人的因素是不容忽视的参数。对我们来说，国情、体制、人员的工作习惯等都应该做具体分析。只是因为目前我们还十分缺乏软件开发组织工作的实践经验，了解并参考国外的做法是必要的，但无论如何不应简单地搬用。

(1) 组织原则

在建立组织时应注意到以下的原则:

① 尽早落实责任:在软件开发项目工作的开始，就要尽早指定专人负责。使他有权进行管理，并对任务的完成负责。

② 减少接口:开发过程中，人员之间的联系是必不可少的。但一个组织的工作效率是和完成任务中存在的人际联系数目成反比的。

③ 责权均衡:软件经理人员所负的责任不应比委任给他的权力还大。

(2) 组织结构的模式

通常有三种组织结构的模式可供选择:

① 按课题划分的模式(Project format):做法是把软件人员按课题组成小组，小组成员自始至终完成所分配课题的各种任务。他们要负责完成产品的定义、设计、实现、测试、复查、文档编写，

甚至包括维护在内的全过程。

② 按职能划分的模式(functional format)：参加工作的软件人员按任务的工作阶段划分成若干专业小组。要开发的软件产品在每个专业小组完成阶段加工以后沿流水线向下传递。比如，分别建立起计划组、要求分析组、设计组、实现组、系统测试组、质量保证组及维护组。系统定义、项目计划、软件要求说明书等文档资料按工序在各组之间传递。各组人员定期轮换可能必要，为的是减轻每个软件人员因长期做单调的工作而产生的乏味感。这种模式在小组之间的联系形成的接口要比第一种模式多。但这样的组织却有利于软件人员熟悉小组的工作，进而变成这方面的专家。

③ 矩阵形模式(matrix format)：图 8.9 示出了这种模式。矩阵

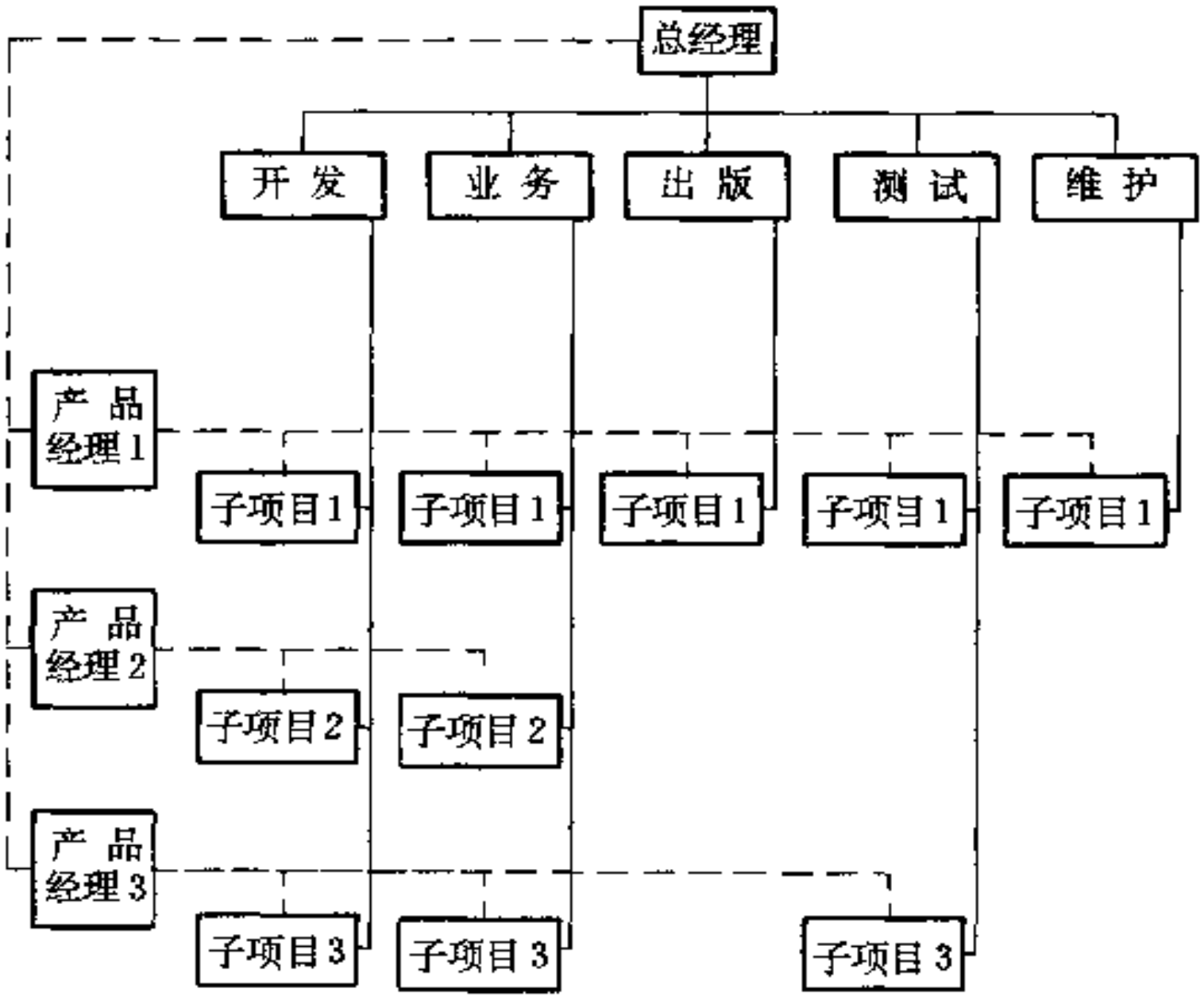


图 8.9　矩阵形组织

形结构实际上是把上述两种模式结合起来。一方面按工作性质，成立了一些专门组(如开发组、业务组、测试组等)；另一方面每一个项目又有它的经理人员负责管理。属于专门组(如测试组)的软件人员，参加某一项目(如第三项产品的研制)中的测试工作。实际上任何一个软件人员都受双重领导(一是专门组，另一是所参加项目的经理)。矩阵形结构的组织具有一些优点：参加专门组的成员可在组内交流各项目工作中取得的经验，这更有利于发挥专业人员的作用。另一方面各个项目也有专人负责，有利于任务的完成。显然，矩阵形结构是一种较好的组织形式。

(3) 程序设计小组的组织

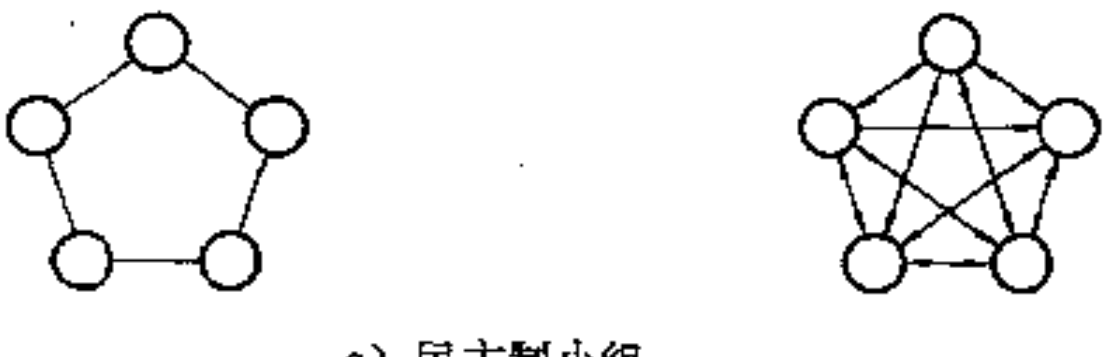
通常认为程序设计工作是按独立方式进行的。程序人员独立地完成任务。但这并不意味着互相之间没有联系。值得注意的是联系的多少和联系的方式与工作效率直接相关。程序设计小组内人数少，比如两人或三人，工作的联系比较简单。但在增加人员数目时，相互之间的联系复杂起来。并且不是按线性关系增长。如果小组内有 n 名成员，组内人际联系的数目为

$$n(n-1)/2$$

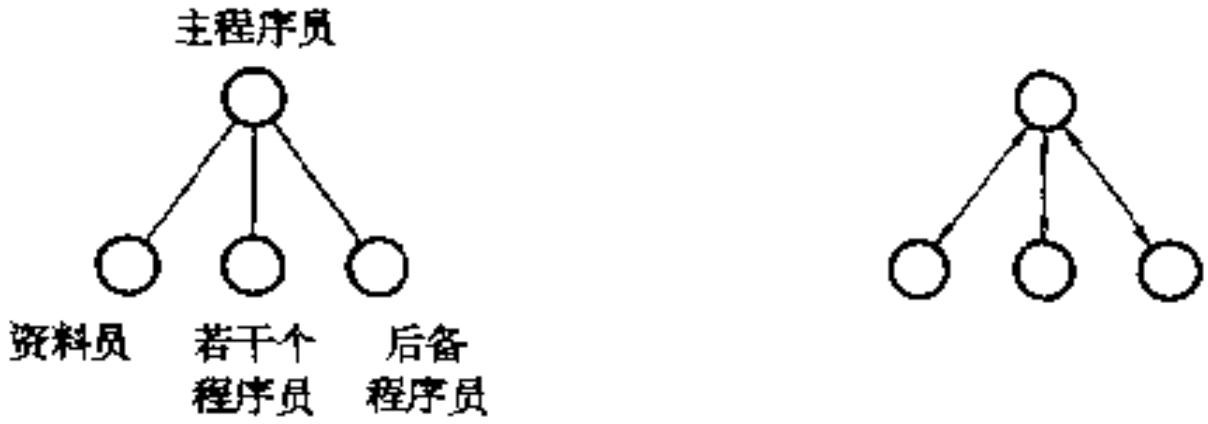
这一关系告诉我们，已经进行的软件项目在任务紧张，延误了进度的情况下，不鼓励增加新的成员给予协助。除非分配给新成员的工作是比较独立的任务，并不需要对原任务有更细致的了解，也没有技术细节的牵连。这样做完全是出自工作效率的考虑。Brooks 说，软件开发工作不同于麦收或采棉。不能把工作量(若干人月)简单地分配给几个软件人员去做。正如一个婴儿是母亲在九个月内养育的，但无论如何也不能让九个母亲在一个月内养育出一个婴儿来。因此，Brooks 认为，在已经延误进度的软件项目中增加新的人员，只会使任务进一步拖延。

小组内部的人员的组织形式对工作也会带来影响。我们看到现有的组织形式有三种：

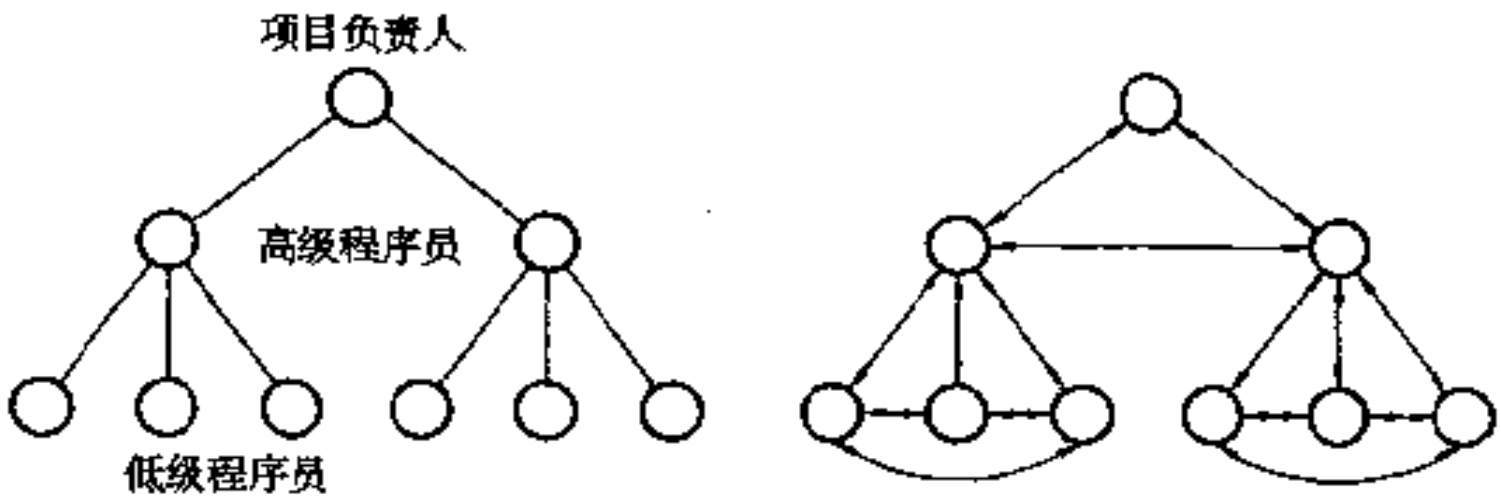
① 民主制小组(democratic team):遇到问题组内成员之间可以平等地互相交换意见(见图 8.10 的 a 图)。工作目标的制定以及作出决定都由全体成员参加。虽然也有一人担任组长,但工作的讨论、成果的检验都公开进行。这种组织形式在讨论时可以充分地听取每个成员的意见,并能互相学习,在组内形成一个良好合作的工作气氛。但有时也会因此而削弱了个人的责任心和必要的权



a) 民主制小组



b) 主程序员制小组



c) 层次式小组

图 8.10 三种不同的小组结构(左三图为结构形式,
右三图为通讯路径)

威作用。有人认为这种组织形式适合于研制周期长，并且难度较大的项目。

日本在发展计算机事业中，组织软件开发大多采用这种形式的开发小组，取得了很好的效果。他们强调发挥每个小组成员的积极性，要求每个成员充分发挥主动精神和协作精神，也创造了一个尊重每个成员的良好工作环境。使得开发小组同时也是质量管理小组。由于小组人员在工作上能够很好地配合，因而做到了较长时间稳定的人员合作关系。这样的小组形式可以避免美国人在“个人自由”口号下，软件人员频繁流动对工作造成的严重干扰。就软件工作的特点来说，这一点十分重要，对我国发展软件产业应该有所启发。

② 主程序员制小组(chief programmer team)：主程序员制的小组设主程序员一人，程序员 3 到 5 人，可能还有资料员和后备咨询人员(参看图 8.10 中的 b 图)。主程序员负责设计并实现项目中的关键部分，对主要的技术问题做出决定，并给程序员分配工作。程序员承担编写代码和文档资料，完成单元测试和调试的工作。资料员负责维护程序清单、设计文档资料、测试计划等。后备程序员为主程序员提供技术咨询，但也做部分分析、设计和实现的工作。此外还有一位行政管理人员协助主程序员处理行政事务。

主程序员制的开发小组突出了主程序员的领导。强调主程序员与程序员的直接联系。总的说简化了人际通讯。这种集中领导的结构能否取得好的效果，很大程度上取决于主程序员的技术水平和管理才能。

美国的软件产业中大多是主程序员制的工作方式。

③ 层次式小组(hierarchical team)：小组中项目负责人(参看图 8.10 的 c 图)给程序员分配任务、参加评审和遍查(review and walkthrough)、掌握工作量并参加技术活动。这种结构只允许必要的人际通讯。比较适用于项目本身就是层次结构状的课题。因为

这样可以把子项目分配给基层小组。例如，具有三个子项目的课题由具有三个基层小组的层次小组完成。基层小组的领导与项目负责人直接联系。通常基层小组成员不超过十人。因而，大型项目需要划分成若干层。

这种层次式小组结构的缺点是，优秀的程序员被提拔到管理岗位上时，小组内失去了好的程序员。但新提拔的管理人员不一定具有良好的管理技能、通讯联络技能或者较强的组织能力。

上述三种组织形式可以灵活运用。例如，较大的软件项目也许是把主程序员小组组织成层次式结构；也许层次小组的领导又是一个民主小组的成员。

8.4 配备人员

怎样合理地配备人员，也是成功地完成软件工程项目的切实

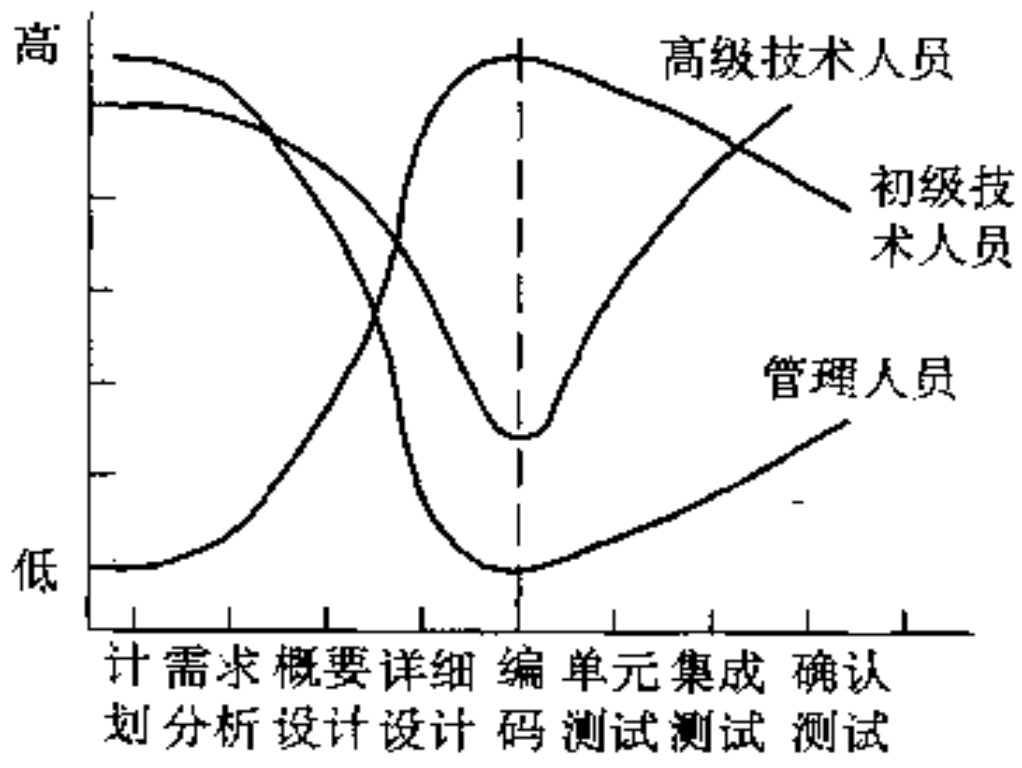


图 8.11 各开发阶段几类人员参与项目的程度

保证。所谓合理地配备人员应包括：按不同阶段适时任用人员，恰当掌握用人标准。

(1) 项目开发各阶段所需人员

在软件开发各阶段中需要不同的人员。图 8.11 给出了较大项目各类人员参与的程度。这个图表明，高级技术人员和管理人员参与程度的变化十分相似，在需求分析和确认测试阶段需要他们作出较多的贡献。另一方面详细设计、编码和单元测试则要求初级技术人员的实际参与。

Boehm 对两个不同规模的软件项目做了统计。一个是具有 32,000 语句行的项目，另一个是具有 128,000 语句行的项目。

统计结果如表 8.3 给出。

表 8.3　各开发阶段投入工作量比例

开发阶段	工作量所占比例(%)	
	32 KDSI*	128 KDSI
计划及需求分析	6	6
概要设计	16	16
详细设计	24	23
编码及单元测试	38	36
系统测试	22	25

* KDSI 为千源指令

这个表只是开发阶段所需人力百分数，并未包括维护阶段，如果拿整个开发与维护阶段所需人力对比，大约是 40：60 或 30：70 甚至 10：90。从这些数字可粗略地看出各个阶段投入人力的情况。

(2) 配备人员的原则

配备软件人员时，我们应注意到三个主要的原则：

① 重质量：事实表明，软件项目是技术性很强的工作，任用少量有实践经验、有能力的人员去完成关键性任务，常常要比使用较多的经验不足的人员更有效。

② 重培训：花力气培养所需的技术人员和管理人员是有效地

解决人员问题的好办法。

③ 双阶梯提升：人员的提升应分别按技术职务和管理职务进行。不能混在一起

(3) 对项目经理人员的要求

软件项目的经理人员是工作的组织者，他的管理能力强弱是项目成败的关键。除去一般的管理要求外，他应具有以下的能力：

① 把用户提出的非技术性要求加以整理提炼，以技术说明书的形式转告给分析员和测试员。

② 能说服用户放弃一些不切实际的要求，以便保证合理的要求得以满足。

③ 能够把表面上似乎无关的要求集中在一起，归结为“需要什么”，“要解决什么问题”。这是一种综合问题的能力。

④ 要懂得心理学，能说服上级领导和用户，让他们理解什么是不实际的要求，但又要使他们毫不勉强，乐于接受，并受到启发。

(4) 评价软件人员的条件

软件工程项目中人的因素越来越受到重视。在评价和任用软件人员时，必须掌握一定的标准。人员的素质优劣常常影响到项目的成败。能否达到以下一些条件是不应忽视的：

① 牢固掌握计算机软件的基本知识和技能。

② 善于分析、综合问题，具有严密的逻辑思维能力。

③ 工作踏实、细致，不靠碰运气，遵循标准和规范，具有严格的科学作风。

④ 工作中表现出有耐心、有毅力、有责任心。

⑤ 善于听取别人意见，善于与周围人员团结协作，建立良好的人际关系。

⑥ 具有良好的书面和口头表达能力。

8.5 指导与检验

指导是软件管理的第四方面工作，其目的是在实施软件工程项目过程中，动员和促进工作人员积极完成分配的任务。实际上，指导也是属于人员管理的范围。是组织好软件工程项目不可缺少的工作。

检验是软件管理的最后一个方面。它是对照计划检查执行情况的过程，同时也是对照软件工程标准或软件工程规范检查实施情况的过程。在发现项目实施与计划或与标准有较大偏离时，应采取措施加以解决。

(1) 指导工作的要点

在指导软件工程项目中需注意到以下几个方面的问题：

① 鼓励：对工作的兴趣和取得的显著成绩常常能够成为推动工作的积极因素。恰当而且及时的鼓励是非常重要的。它可使人们充满信心，勇于继续克服困难，愿意努力进一步提高工作效率，迎接新任务的挑战。

② 引导：通常人们愿意追随那些能够体谅个人要求或实际困难的领导。高明的领导人应能注意到这些，并能巧妙地把个人的要求和目标与项目工作的整体目标结合起来，至少应能做到在一定程度上的协调，而不应眼看着矛盾的存在和发展，以致影响工作的开展。对于合适的人员应让他们喜欢在你这里工作，不愿离去。须知，大幅度的人员调整肯定是非常有害的。它会带来许多实际问题。即使是人员的临时观念也都要使项目付出不可见的代价，因而蒙受无形的损失。

③ 通讯：在软件工程项目中充满了人际通讯联系。必要的通讯联络肯定是不可少的，但实践表明，工作效率是通讯量的函数。如果人际通讯数量过大，会使软件生产效率迅速下降。

(2) 检验管理的要点

在检验管理中应该注意到：

① 重大偏离：在软件工程项目实施过程中，必须注意发现工作的开展与已制定的计划之间，或与需遵循的标准（或规范）之间的重大偏离。遇有这种情况应及时向管理部门报告，并采取相应措施给予适当的处置。

② 选定标准：检验管理需要事先确定应该遵循的标准（或规范），使得软件项目的工作进展可以用某些客观、精确且有实际意义的标准加以衡量。

③ 特殊情况：任何事务在一般规律之外都会存在一些特殊情况，管理人员必须把注意力集中在软件项目实施的一些特殊情况上，认真分析其中的一些特殊问题，加以解决。

（3）检验管理的工作范围

检验管理在软件工程项目中可能涉及到：

① 质量管理：包括明确度量软件质量的因素和标准、决定质量管理的方法和工具以及实施质量管理的组织形式。

② 进度管理：检验进度计划执行的情况。

③ 成本管理：度量并控制软件项目的开销。

④ 文档管理：检验文档编写是否符合要求。

⑤ 配置管理：检验软件配置（下节将进一步讨论）。

（4）软件工程项目中人的因素

软件产品是人们大量智力 劳动的结晶，软件工程项目能否获得成功，人的因素在其中所起的作用比其它任何工程项目都更加突出。有人总结和分析了 60 个软件项目的数据，对影响软件生产率的因素进行了研究（见 B. W. Boehm 所著“软件工程经济学”一书）。结果表明，软件人员能力发挥得怎样对软件生产率的影响极大。从图 8.12 中可看出，在与软件成本相关的 14 项属性中，软件人员能力一项是最大影响因素。它表明，如果在软件项目中能充分发挥软件人员的积极性，使他们的才能得到尽量的施展，软件

生产率(以单位时间内开发出源程序的平均行数计算)最高可提高四倍多。

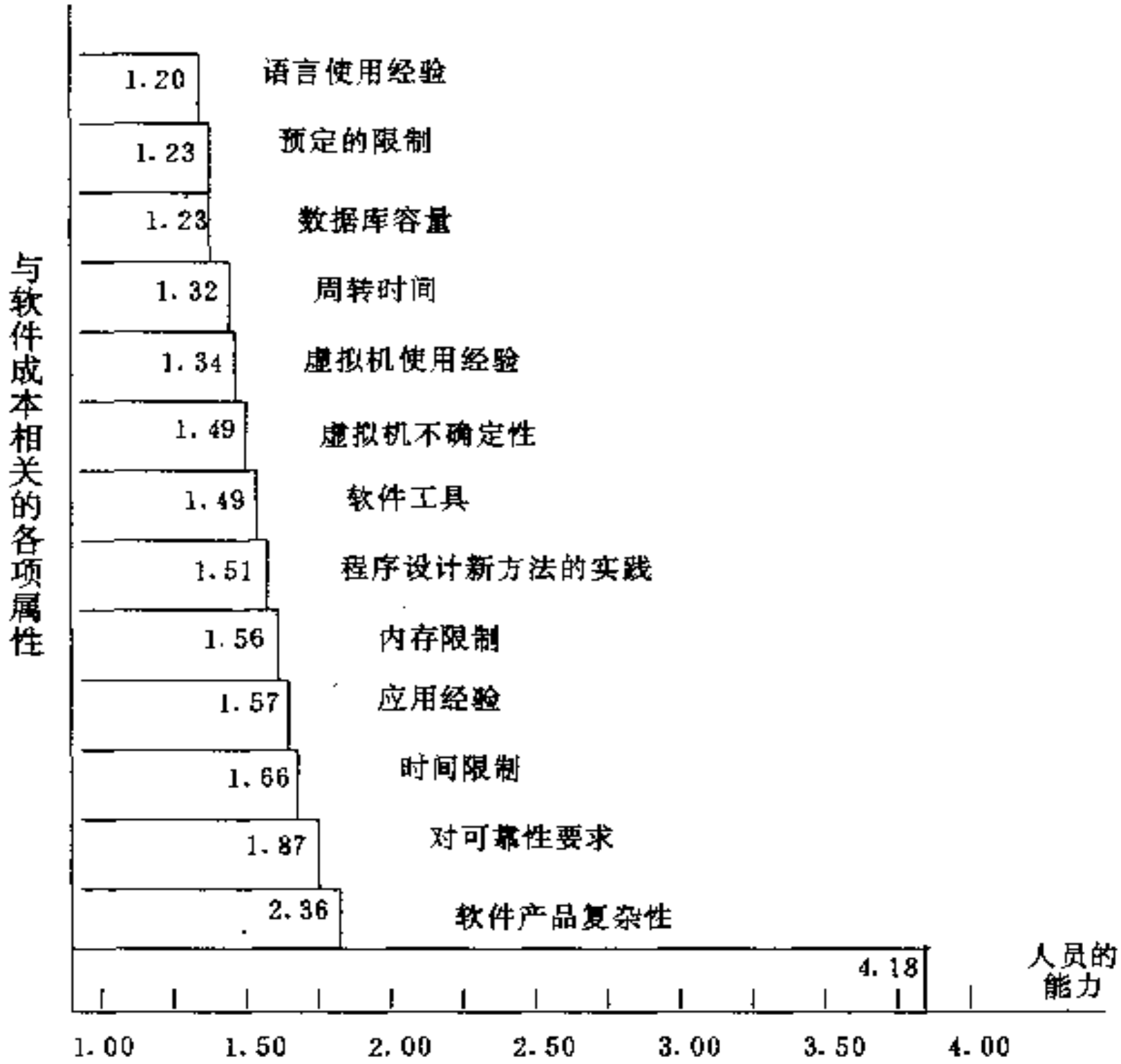


图8.12　与软件成本相关的属性对生产率的影响

由此足见人员因素的重要性。

著名软件工程专家 Tom DeMarco 最近著书(Tom DeMarco and Timothy Lister, Peopleware: Productive Projects and Teams, Dorset House Publishing Co. Inc., 1987)专题讨论了软件生产中人的因素问题。积 30 年软件项目管理的经验，他认为软件工程项目中对于

人员的管理问题不能象其它事物那样简单地划分、机械地对待。

从 1977 年以来，作者注意考察了许多软件开发项目的实际情况，特别注意到项目的规模、成本、缺陷、加快开发的因素以及执行进度计划中的种种问题。他们积累了 500 个项目开发过程的数据，从中发现大约 15%的项目失败了，有的是一开始就被撤销，有的中途流产，有的推迟了进度，有的成果不能投入使用，而且愈是大一些的项目就愈糟。究其原因，绝大多数失败的项目竟找不出一个可以说得出口的技术障碍；障碍却来自人员之间的联系问题、人员的任用问题、对上级或对顾主失望、工作缺乏动力或缺乏高额工程维持费 等等。这些人际关系的问题的解决可归结为“软件项目社会学”。

关于软件人员的办公环境，有许多因素影响着软件工作的效率。DeMarco 曾于 1984—1986 年在 62 个公司的 600 名软件人员中进行编码和测试竞赛活动，并对竞赛结果作了统计分析。结果表明，除对语言的熟悉程度、工作年限、工资收入等因素外，环境因素起着很大的作用。良好的办公环境可保证软件人员高质量完成任务。这里说的办公环境指的是每个软件人员的办公室工作面积、办公环境安静程度、专用程度、电话干扰程度，工作时间内外找人次数等。DeMarco 说，你如果是一名项目管理人员，你为软件人员安排了任务，提供了工作条件，而对工作环境所带来的影响估计不足，你还是应该承担责任。

8.6 软件配置管理

(1) 配置管理的任务

任何软件在它的生存期的各个阶段都要产生一些文档。包括各种计划、分析、设计、编码、测试、运行以及维护方面的资料，名目繁多，其数量不下几十种。同时我们还注意到，在某些阶段，由于特定的原因，需要对其中一些做必要的修改和更动。这种变动势必

牵连到其它资料也要随之变动。另一方面，修改后的资料固然重要，但修改前的资料还会有一定的参考价值，尚不能立刻丢掉。这样各阶段的资料又加上多种版本，构成了十分复杂的关系。如果没有一个科学的管理办法，很容易造成混乱。这就为软件的管理提出了新的课题。

其实，配置(Configuration)的概念最早用于硬件，比如，我们说到某个计算机系统的配置时，指的是，该计算机系统的中央处理机如何，与它配合工作的一套外部设备，如磁盘、磁带、打印机、绘图仪等数量多少，性能如何。这时，人们往往把软件看做一个整体。近年来，在积累了软件开发和软件运行、维护的实践经验以后，逐渐认识到软件的复杂性，开始感到有必要把配置的概念应用于软件，并且需要十分重视软件配置的管理工作。

一个软件在某一特定时刻的配置可以看作为该软件物理表示在此时的瞬时快照。这里所说的物理表示可有两种形式：

① 不可直接执行的资料，如需求说明、设计说明、程序清单、测试数据等。

② 可直接执行的资料，如存入计算机的机器可读代码、数据库信息等。

我们把软件配置管理的对象称为软件配置项，它包括：

系统规格说明书	测试报告
软件项目开发计划	操作手册
软件需求规格说明书	用户手册正式稿
可供使用的原型	可直接运行的目标码程序
用户手册初稿	软件问题报告
概要设计规格说明书	维护请求
详细设计规格说明书	工程变更通知
源程序清单	软件工程标准
测试计划	项目开发总结

另一方面，我们又可把这些资料分成两类：

① 在开发过程中得到的一些仅供开发人员使用的资料。

② 对外公开，可提供用户的资料。在图 8.13 中，我们看到可提供用户的软件配置。

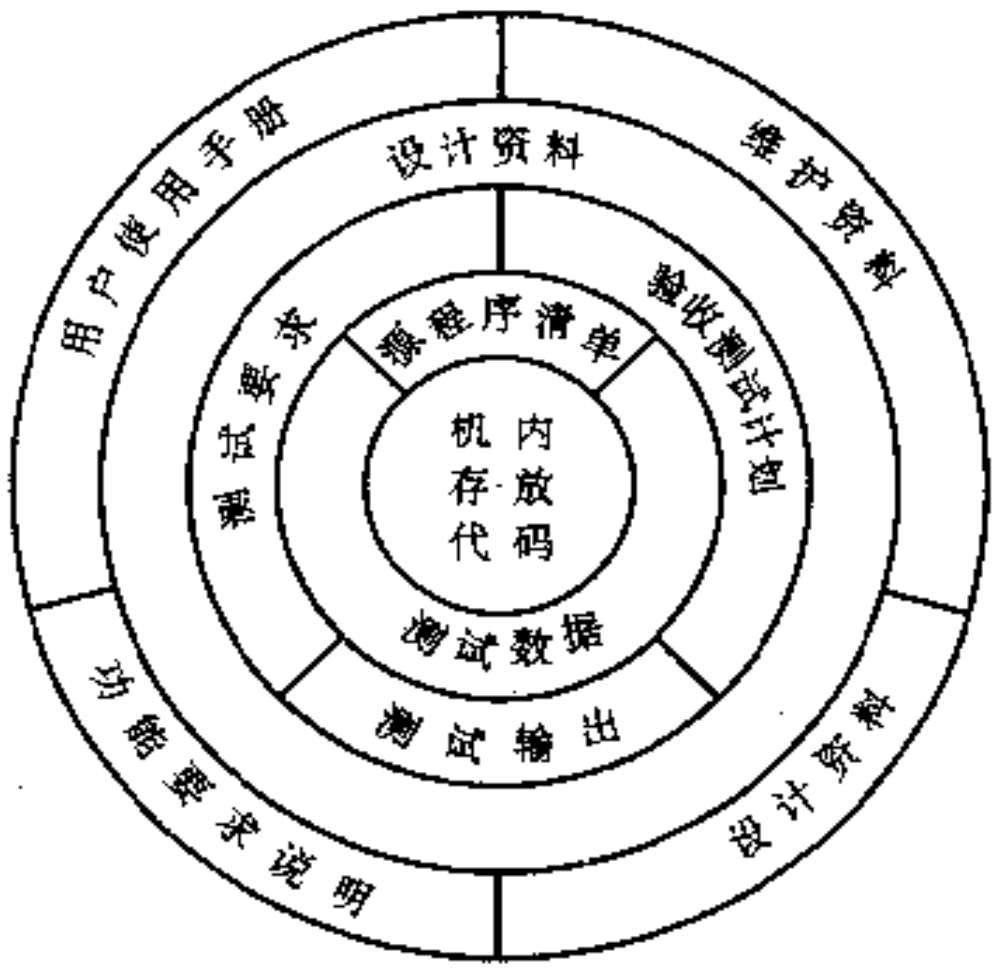


图 8.13　交付用户的软件配置

软件配置管理的任务包括：

① 建立软件文档的书写标准，规定一个文档标识的方案。

② 进行变更控制，审定并且记录各种软件配置的变更。

③ 对已完成的文档能够准确地跟踪和受控存取

④ 对软件配置进行不断的检查。

(2) 配置标识

软件配置实际上是一个动态的概念。一方面随着软件生存期的向前推进，文档逐渐增多；另一方面又会有新的更动出现，形成多种版本。如果我们把某一软件特定时刻的配置看成瞬间照片，那么就整个生存期来说，它的配置就是不断演变的一部电影胶卷。

为了方便对软件配置各个片段进行处理，使其不致造成混乱，首先应给整个胶卷和其中的每一镜头贴上可供识别的标签。因此，配置标识的任务是确定一个文档的组织结构，这种标识的办法应当是容易理解的，也能作延伸性推测的。并且，它应该能够应付各种资料的修改变更，在发生变更时，能够跟踪变更。为便于掌握变更的情况，应把变更与相关的因素联系起来，很快地搞清谁在什么时间，为什么做这个变更，而且当时是怎样做的变更。

具体说来，我们需要一组很容易被人接受的软件资料标签号。比如，

TCC-004-M-05-9/84

表示

清华计算中心(Tsinghua Computer Center)第 004 号课题的维护手册(Maintenance manual)第 05 号版本，于 1984 年 9 月完成。

软件配置标识的一般形式是

×××-YYY-Z-RL-NNN

其中，×××指明了某软件课题的组成标识(Component identifier)

YYY 指明了课题，是课题标识(Project identifier)

Z 是配置分类标识(item identifier)，如 Z 可以是：

P	计划
R	要求说明
D	设计资料
S	源程序清单
T	测试资料
U	用户手册
I	安装指南
M	维护手册
RL	表示修改更动的次数
NNN	是属性码，用以表达配置的重要属性，如日期。

建立了配置标识方案以后，往往需要用数据库来作储存和管理的支持。

(3) 基线的概念与变更控制

软件工程项目中修改和更动总是不可避免的。用户要求修改已提出的需求；软件开发人员要修改技术方案；管理人员提出要修改实施计划等等。随着时间的前进，人们对事物的认识进入了更深的层次，合理的修改和更动总是朝着最终目标前进了一步。我们不能只是从消极的方面，简单地拒绝一切合理的变更。问题是如何进行控制和管理，选择那些变更是合理的，变更的理由是充分的，变更的作法也是可行的。所谓变更控制包含两个方面：建立控制点和建立报告与审查制度。

① 基线(baseline)

基线是软件生命期中各开发阶段末尾的特定点，也称为里程碑(milestone)。它的作用是把各阶段工作的划分更加明确化，使本来连续的工作在这些点上断开，使之便于检验和肯定阶段成果。例如，明确规定不允许跨越里程碑修改另一阶段的文档。

这里以餐厅和厨房的服务为例说明基线对于变更控制的作用。餐厅和厨房各自的职能不言而喻。如果从餐厅到厨房设有两个单向门，一个入口门，一个出口门。某服务员在厨房发现客人定的菜与所供的菜不符，他需要找客人确认后通知厨师改正，要经过以下步骤：

- 服务员从出口出来找到客人证实菜单并致歉；
- 服务员从入口进入，向厨师解释清楚；
- 厨师改作另一菜。

我们从这一例子中发现，两个门起了控制作用：要和客人谈得经过出口门到餐厅来；要让厨师改菜得经过入口门进入厨房。

在设置基线以前，就如同餐厅和厨房联通，没有门存在，也就没有控制作用。然而基线一旦建立起来，要处理跨阶段的工作，

就如同必须经过上述一些特定的严格步骤那样，要经过评价、确认，在肯定这一变更是必要时，才施行。图 8.14 表明了软件开发各阶段的基线。以需 求基线为例，用户对需求的认识是逐渐深入的，当他的初步需求得到满足后，常常会提出进一步的需求。然而，他并不了解需求的变更对软件开发会带来多大影响。软件开发人员必须正确对待用户的变更要求。首先，一开始确定需求时应在全面调研用户业务领域的基础上，采取慎重的态度。对于今后可能的变更做出估计，力争在需求规格说明书中考虑到可能的变更。另一方面，完成需求分析进入设计阶段以后，如果再提出对需求的变更，就不那么容易了。这时需求基线已将用户需求“冻结”了，如用户坚持新的需求，只好在完成本项目开发以后，作为新项目的需求考虑了。

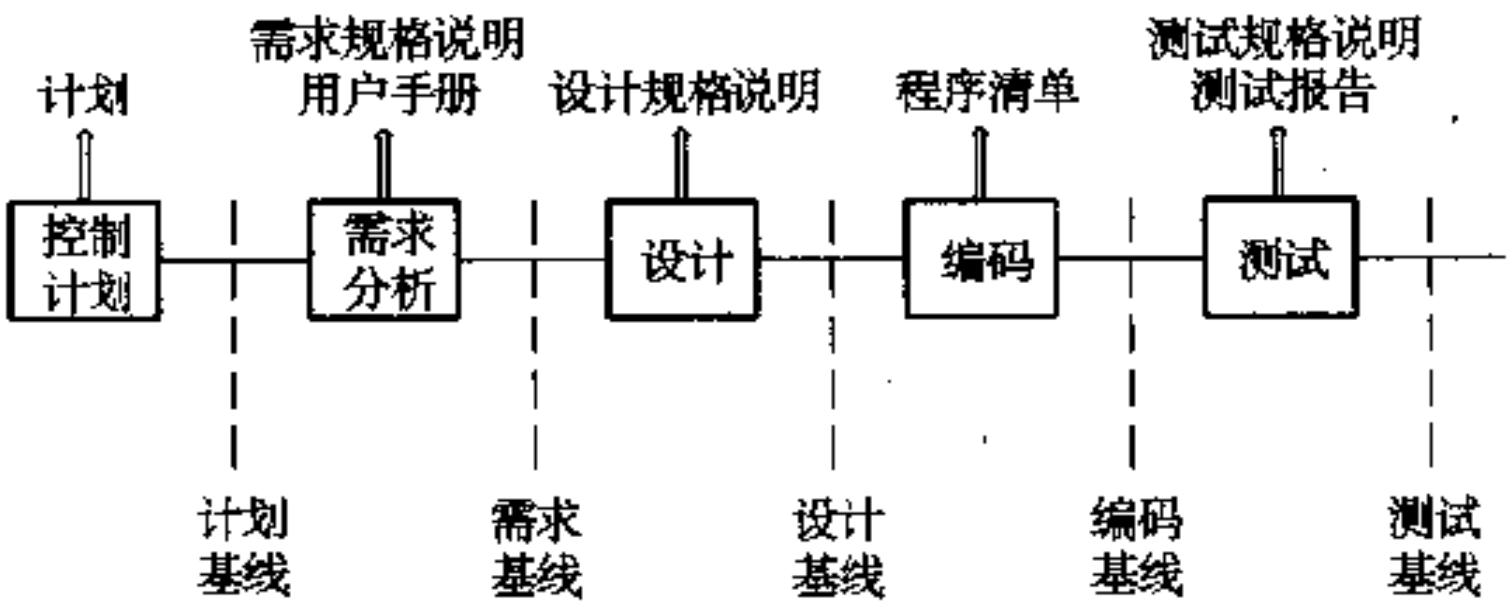


图 8.14　软件配置基线

② 变更报告与审查

为严格、慎重地对待变更请求，建立变更报告与审查制度是完全必要的。首先，变更要求者应向管理部门提出书面的变更请求报告。详细申明变更的理由、变更的方案以及变更的影响范围等。软件的变更通常有三种不同的情况：

• 软件开发过程中提出的变更：这种变更往往限于开发机构内

部，较少涉及、甚至完全不涉及到用户。因此，只需按照一定的审查制度进行。

· 软件开发中因突然事件需作的紧急变更：这可能影响到用户。

· 进入运行维护阶段的变更：这种变更涉及的面更广，必须慎重进行。

图 8.15 给出了变更控制的过程。在接收变更请求以后，需对技术可行性、时间、经费以及可能波及的影响进行分析和估计，并将所得到的变更报告提交变更控制委员会(Change Control Board)

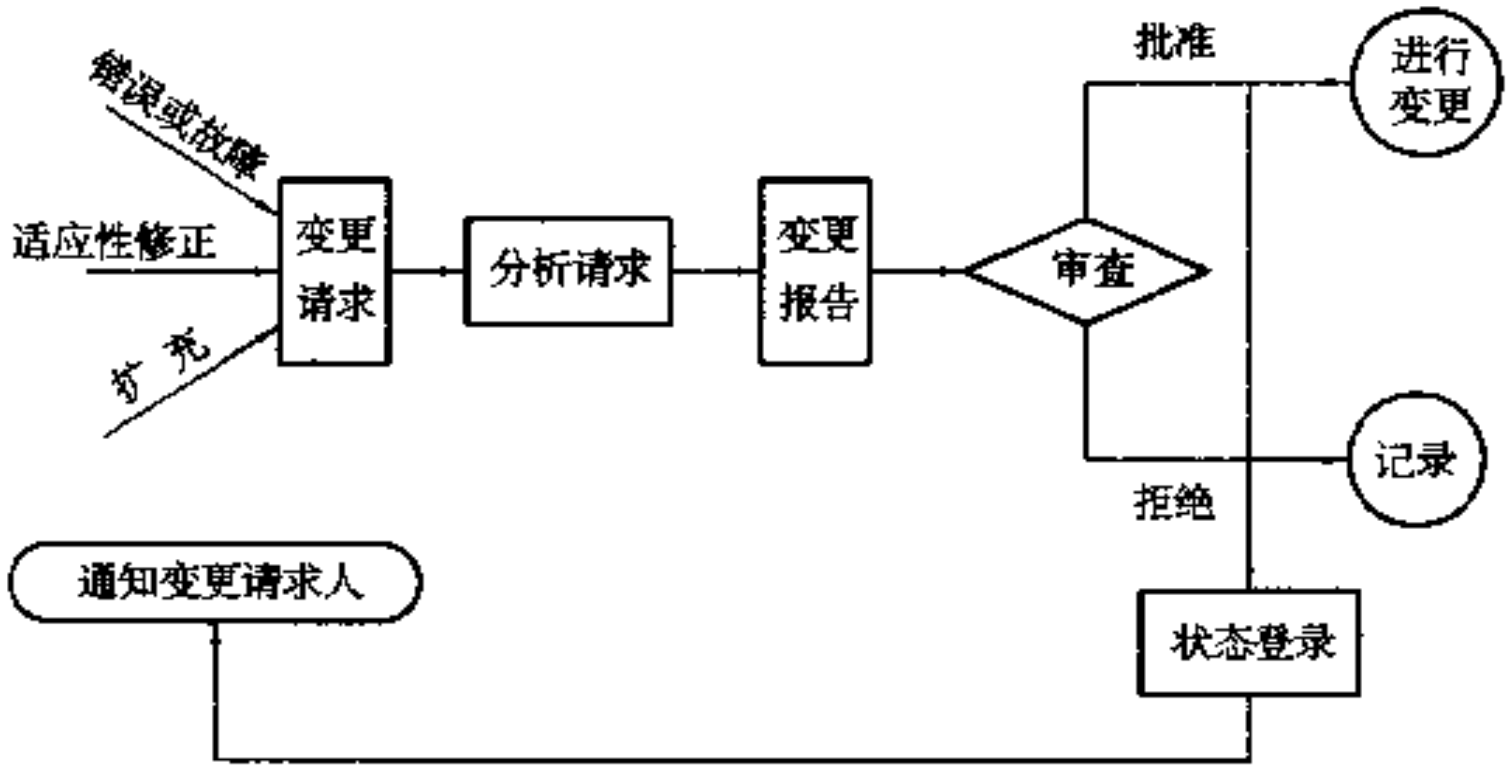


图 8.15 变更控制过程

进行审查，根据情况分别作出批准与拒绝两种不同的处理，并通知变更请求人。

(4) 配置状态登录(Configuration status accounting)

随着软件开发工作的进展，除去产生了一些成果性的资料外，还有一些辅助性的资料，例如软件变更请求，问题报告，变更报告等。也许有些报告经过审批付诸实施，有些则被拒绝，为了清楚、及时地记载这些变化多端的活动，而不致到后期造成贻误，需要对开发的过程作出系统的记录，以反映开发活动的历史情况。这

就是配置状态登录的任务。其实，这种登录主要根据变更控制小组会议的记录。因为记录中已能很好地反映出所需要的主要信息。这种登录工作在信息量足够大时，手工的办法已不能负担，需要有计算机协助来完成，即以数据库的形式给予支持。这就极大地方便了使用，不仅可以高效地登录，还可以高效地查询和输出所需的信息或报告。

(5) 配置审计(Configuration audit)

任何一个软件从开发到投入使用，要经历许多阶段。也会有不同的人员参与。如何确保软件的要求在后期工作中得到不失真地体现。或者说，任何后期工作中的软件产品能否正确地反映用户提出的软件要求。这个重要问题称为软件的完整性。

配置检查的目的就是要证实整个软件生存期中各项产品在技术上和管理上的完整性。同时，还要确保任何文档资料的内容更动都不超出当初确定的软件要求范围。使得我们的软件配置具有良好的可跟踪性。这也正是软件变更控制委员会掌握配置、进行审批的先决条件。

图 8.16 把配置管理的四个方面的任务和相互关系体现了出来。从中可以看出配置管理就是要对以上所提到的四方面任务提供必要的指导和监督。

弄清楚软件配置管理的含意和任务以后，自然就会理解它在软件管理中的重要地位。我们可以进一步认识到，作好软件配置管理就能很好地避免出现以下一些情况：

① 由于软件的完整性差、可跟踪性差，造成可靠性差而使软件的维护成本很高。

② 想要改进现有的软件以满足新的要求，但由于没有软件配置的支持无法进行。

③ 使用过时的文档或找不到需要的资料。

由此可以看到，软件配置管理在软件项目中是不可少的。

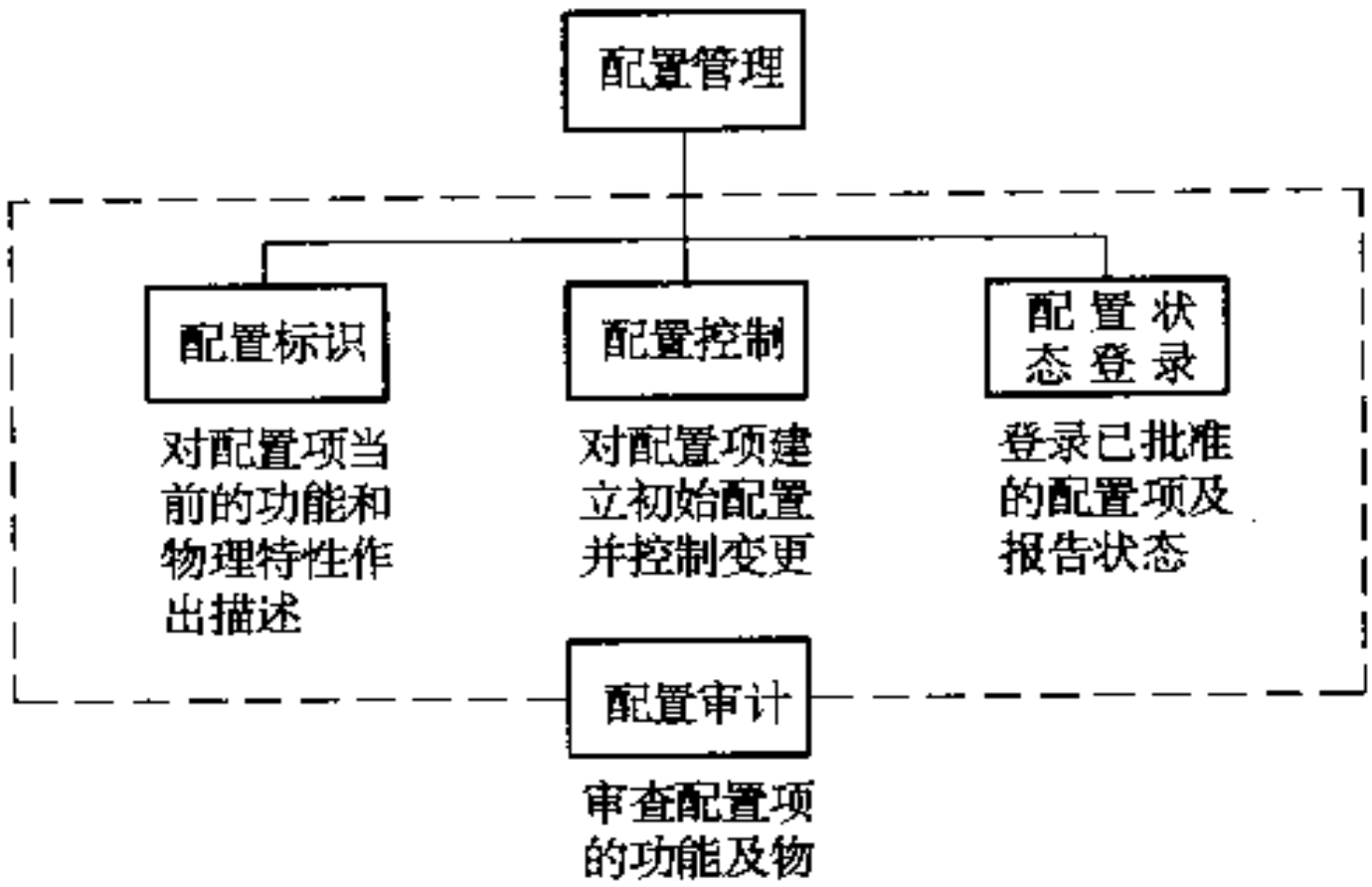


图 8.16　配置管理的任务

8.7　软件成本估算

软件产品凝聚了软件人员的大量劳动。这些劳动的价值理所当然地应该在它的成本中体现出来，但软件产品的成本估算是软件工程中最难解决的问题之一。特别是在软件产品的开发工作尚未开始以前，所需投入的工作量也无法准确地估计，这种情况给合同的签订带来了一定的困难。为缓和这一矛盾，有的公司采用“三段估量”的方法，即在开发的初期定出初步估算：在要求分析阶段，多少掌握了一些估价的资料，便给出修正估算；在初步设计完成以后给出最后估算。也有人索性采取按开发阶段签订合同的办法，以减小矛盾。不过以上这些做法都没有从正面去解决问题，而是回避了这个难题。

近年来，在软件成本估算方面出现了许多研究工作。这些研究大多从分析与软件成本相关的因素入手。

(1) 影响软件产品成本的主要因素

① 软件人员的业务能力。由于每个软件人员的素质、经验、掌握的知识不同，在工作中的表现有着很大差别。

② 软件产品的复杂程度和规模。按软件产品的不同复杂性，可分为三类：

应用程序——用高级语言写的科学计算、数据处理等的用户程序，其复杂性最低，而生产率最高。

实用程序——用系统程序设计语言或汇编语言写出的汇编程序、编译程序、连接编辑程序、输入装入程序等。

系统程序——用汇编语言或系统程序设计语言写出的操作系统、实时处理控制系统等，其复杂性最高，而生产率最低。

粗略地统计数字表明，系统程序——实用程序——应用程序之间的生产率和复杂性的比例关系是：

	系统程序	实用程序	应用程序
生产率	1	5—10	25—100
复杂性	9	3	1

Boehm 进一步给出了这三种软件产品的程序规模与工作量的关系式。

若程序的规模以完成项目后交付的源程序每千条指令(语句)为单位，记为 KDSI。程序工作量以人-月为单位，记为 PM。

其关系式分别为

应用程序　　PM＝2.4＊(KDSI)＊＊1.05

实用程序　　PM＝3.0＊(KDSI)＊＊1.12

系统程序　　PM＝3.6＊(KDSI)＊＊1.20

例如，同样是开发 60,000 行的程序，其工作量不同，所需投

入的人-月数大体上是 1：1.7：2.8(参看图 8.17)。

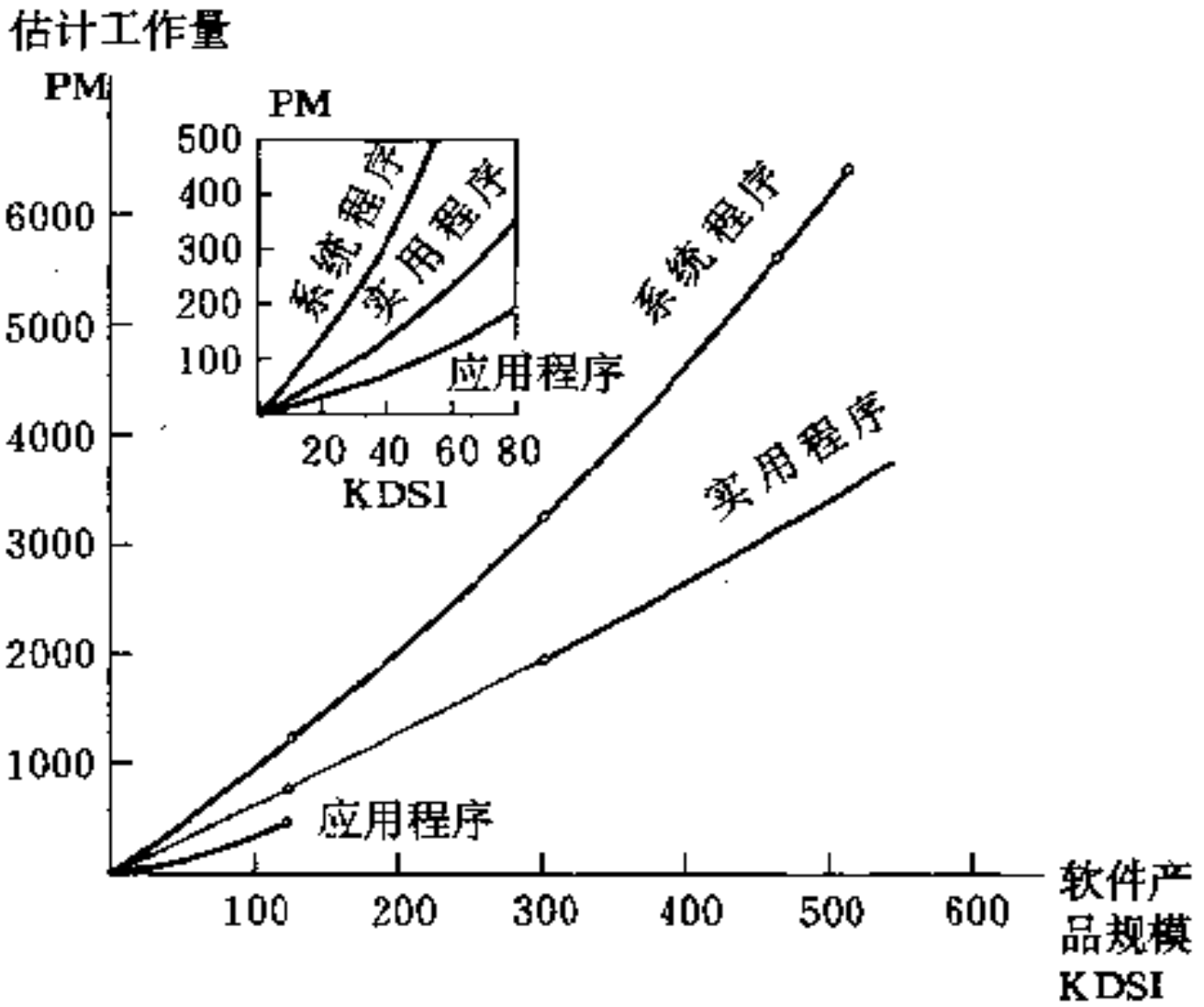


图 8.17　工作量估计曲线

③ 开发软件所用时间。若以 TDEV 表示开发软件所需时间，Boehm 给出的计算为

应用程序　　TDEV＝2.5＊(PM)＊＊0.38

实用程序　　TDEV＝2.5＊(PM)＊＊0.35

系统程序　　TDEV＝2.5＊(PM)＊＊0.32

值得注意的是，同样大小的程序，以上三种类型问题所需的开发时间相同。例如，60KDSI 的程序都需 18 个月。同时，如果我们把开发时间拉长或缩短，都需要更多的工作量。称此时间为最佳开发时间。

在研究了 63 个软件项目的开发工作以后，Boehm 得出结论：软件项目开发所用的时间有一个极限值，在此极限值以外，若想用增加人员和设备的办法来缩短开发时间是不可能做到的。这个

极限值即为上述最佳开发时间的75%。

④ 可靠性。软件的可靠性应定义为一个程序在给定的条件下，按指定的时间完成预定职能的概率，它和程序的精确度、健壮性(robustness)、完整性与一致性有关。

软件的可靠性是在计划阶段，考虑到软件失效的代价而确定下来的。这种失效的情况各有不同，轻者也许只是给用户带来一些不便，重者也许会造成巨大的经济损失，甚至冒着人命关天的风险。根据不同的可靠性要求，对软件开发的工作量有不同的估计。表8.4给出五类软件可靠性相应的开发工作量系数。

表 8.4　和软件可靠性相关的开发工作量系数

可靠性类型	失效的影响	开发工作量系数
很低	不便使用	0.75
低	其损失易于挽回	0.88
一般	挽回损失有一定困难	1.00
高	很大的经济损失	1.15
很高	涉及到人的生命安全	1.40

⑤ 软件开发的技术水平。软件开发项目的技术水平与所用的程序设计语言、程序设计技术和软件工具等因素有关。很显然，用高级语言写出的程序能扩展成更大篇幅的机器语言程序。使用高级语言要比用汇编语言写程序，要提高生产率5至10倍。各种高级语言之间也有很大差别。现代程序设计语言，比如Ada，还可提供一些其它手段，而使软件的生产率和可靠性均有提高。例如，Ada语言中有很强的类型检查设施、数据抽象、单独编译、例外处理、中断处理和并行机构等。如果程序员在软件开发中必须熟悉一套全新的开发方法和工具，或者所用的计算机和软件并行地开

发，甚至用机受到限制，其生产率自然受到影响。而生产率是和软件成本直接相关的。

现代程序设计技术包括，使用系统分析和系统设计的方法、结构化设计方法、遍查和评审、结构程序设计、系统测试和程序库。

软件工具则更广泛，从初等的工具，如汇编程序、基本的调试手段到编译程序、连接编辑程序、交互式文本编辑程序、数据库管理系统、语言处理程序、要求说明分析程序(requirements specification analyzers)以及包括配置管理和自动验证工具在内的软件开发环境。

使用软件工具和现代程序设计技术对软件开发工作量的影响系数在表 8.5 中给出。

表 8.5　现代程序设计技术和软件工具的使用对软件开发工作量的影响系数

	使用情况	开发工作量系数
现代程序设计技术	未采用	1.24
	充分使用	0.82
软　件　工　具	仅用基本工具	1.24
	使用先进的开发工具	0.83

(2) 软件成本估算方法

多数软件开发部门对软件的成本估算是基于以前的实践经验的。过去有关成本的资料，对当前估算很有用。自然，收集当前的数据对未来的工作是有帮助的。

通常采用两种估价方法：

1. 自顶向下估价：其方法是首先对系统估算，然后再考虑子系统。在对系统估价时，涉及到开发该软件系统所需要的资源、人

员、配置管理、质量保证、系统安装、用户培训及资料等方面的成本。

2. 自底向上估算：首先估算系统中每个模块或每个子系统的成本，然后综合成整体的成本。

前一方法的优点是系统一级的因素考虑得多，但可能忽略一些模块中的技术因素，如配置管理、质量控制等方面。最好的做法是两种方法分别使用，然后进行比较和迭代，在消除差别中求得较理想的结果。

(3) 迪菲(Delphi)成本估算法

如前所述，对软件进行成本估算时，以前的经验是重要的依据。因此估算时必须依靠有经验的专家。但是即使是专家也很难完全避免主观片面性，为解决这一矛盾，建议采用 1948 年兰德公司提出的迪菲方法。

该方法的主要思想是，让多位专家分别研究系统定义文件，然后背靠背地单独进行成本估计；在综合各个意见的基础上，让专家们重新考虑，进行新的一轮估计，这样的估计可能进行多次，直至意见相近或者一致。在各轮估计进行时不互相交换意见，由协调人负责综合意见，或者提供参考信息。

(4) COCOMO 成本推断模型

Boehm 建立的自底向上成本估算方法称为成本推断模型 COCOMO——the Constructive Cost Model。其中考虑了多种工作量系数，包括产品属性、计算机属性、人员属性和软件项目属性。各系数的取值范围见表 8.6。这个模型是在研究了 63 个软件开发项目以后，并且使用了迪菲成本估算法所取得的数据得出的。上面提到的几个概念：

交付的源程序指令数	DSI
程序工作量(人-月)	PM
开发所用时间	TDEV

表 8.6　COCOMO 工作量系数

工作量系数	取值范围
产品属性	
要求的可靠性	0.75 至 1.40
数据库规模	0.94 至 1.16
产品复杂性	0.70 至 1.65
计算机属性	
执行时间限制	1.00 至 1.66
主存限制	1.00 至 1.56
虚拟机易失性	0.87 至 1.30
计算机解题周期	0.87 至 1.15
人员属性	
分析员能力	1.46 至 0.71
程序员能力	1.42 至 0.70
实践经验	1.29 至 0.82
使用虚拟机经验	1.21 至 0.90
使用程序设计语言经验	1.14 至 0.95
项目属性	
使用现代程序设计方法	1.24 至 0.82
使用软件工具	1.24 至 0.83
要求的开发进度	1.23 至 1.10

在这里仍然适用。

COCOMO 还做了许多假定，例如，所考虑的项目属于中小型问题(2K 至 32KDSI)；软件人员熟悉项目所属的应用领域；只考虑从设计到验收测试的成本，但包括文档资料和评审的成本；DSI 包括作业控制语句和源程序，但不包括注释语句和未经修改的实用子程序；1(人-月)是指 52 个(程序员-小时)等等。此外还有一些有关软件项目性质的假设。

Boehm 给 出的成本估算例子是商用微型机远程通讯的嵌入

式软件，规模为 10 KDSI。

按公式计算：PM＝2.8(10)＊＊1.20＝44.4

TDEV＝2.5(44.4)＊＊0.32＝8.4

这个问题的工作量系数取值见表 8.7，这些工作量系数的乘积就是工作量调整因子，其值为 1.17。调整后的工作量和开发时间分别为 51.9(人-月)和 8.8 月。

表 8.7　嵌入式远程通讯项目的工作量系数取值

工作量系数	系　数　取　值　的　依　据		取值
可靠性	只用于局部地区、恢复问题不严重	（额定值）	1.00
数据库	20K 字节	（低）	0.94
复杂性	远程通讯处理	（很高）	1.30
定　时	使用 70%的处理时间	（高）	1.10
存　储	64K 中使用 45K	（高）	1.06
机　器	商用微处理机	（额定值）	1.00
周转时间	平均二小时	（额定值）	1.00
分析员	优秀人员	（高）	0.86
程序员	优秀人员	（高）	0.86
经　验	远程通讯工作三年	（额定值）	1.00
经　验	微型机工作六个月	（低）	1.10
经　验	使用语言 12 个月	（额定值）	1.00
实　践	使用现代技术一年以上	（高）	0.91
工　具	基本的微机软件	（低）	1.10
工　期	九个月，估计 8.4 个月	（额定值）	1.00
调整因子＝1.17			

如果程序员和分析员的成本按每月 6 千美元。那么，该项目开发人员的总成本为

(51.9 人-月)＊(6,000 元/人-月)＝311,400 元

软件成本估算是一个很复杂的问题，它所涉及的因素实在太多，所以很难找到一种既准确又简便的方法，不同的估算方法所得的结果可能相差很大。

近年来，由于硬件成本的急剧跌落，软件成本所占的比例越来越高。到现在整个计算机系统成本的 90%以上属于软件。这一事实促使人们更加重视软件成本估算和计价的研究。Boehm 的工作是很有影响的，有兴趣的读者可参阅《软件工程经济学》一书。

第九章　软件工程标准和软件产品文档编制

传统的工业交通部门由于已经积累了百年以上的实践经验，今天所实行的标准化生产的必要性，不会有人提出疑问。实际上，在这些领域里，标准化对生产、流通、运行等方面带来的巨大效益，人们早已习以为常。怎能想象：一台机床维修时，找不到适用的螺丝钉；买到的家用电器因设计电压未考虑到常用的电源电压而无法接电；或是拍发的电报，因对方不识电码而译不出电文等等。在这些方面的标准化问题已成为尽人皆知的常识了。然而，在计算机软件领域中，要不要推行标准化，人们的认识还没有达到应有的高度。至今仍然有人主张，不必对软件人员"限制"太多。对于一名投身于软件产业的工作人员来说，掌握软件工程的知识是必要的，树立对软件工程标准化的正确认识也同样是非常必要的。今天，作为一个产业部门，软件行业已逐步成长壮大起来，它的支柱学科——软件工程已经逐渐成熟，软件产品已进入市场。软件成果的商品化为软件工程标准化创造了必要前提。

另一方面，我们注意到，组织计算机硬件工程虽然也要完成设计和使用资料，但比较起来软件工程的文件资料——文档在整个软件生存期中的地位和作用显得更加重要和突出了。文档作为软件产品的主要形式，集中体现了开发人员的大量劳动。没有文档的软件也就不成其为软件了。在整个计算机系统的工作中对于文档的重要性，无论怎么强调都不过分。因为在实践中文档的编制和使用仍然存在许多矛盾，例如，用户常常手中拿着文档，抱怨它不好用，软件开发人员则不愿花功夫去认真编写文档，认为

不如编写程序来得痛快。究竟文档工作应包括些什么？它有什么特点？怎样才能编出好的文档？本章准备针对上述软件工程标准化问题和软件文档编制问题作一些简要介绍，目的是使读者对它们有个初步的了解。

9.1 什么是软件工程标准化

人们社会生活离不开交往，在交往中最先遇到和首先要解决的是通讯工具——语言文字问题，计算机问世以后，同样是语言问题。人要和计算机打交道，需要程序设计语言，这种语言不仅应让计算机理解，而且还应让别人看懂，使其成为人际交往的工具。程序设计语言的标准化最早提到日程上来。60 年代程序设计语言蓬勃发展，出现了名目繁多的语言，这对于推动计算机语言的发展无疑有着重要作用。但同时也带来许多麻烦。即使同一种语言，由于在不同型号的计算机上实现时，作了不同程度的修改和变动，形成了这一语言的种种“方言”，为编写出程序的交流设置了障碍。制定标准化程序设计语言，为某一程序设计语言规定若干个标准子集，对于语言的实现者和用户都带来了很大方便。

随着软件工程学科的发展，人们对计算机软件的认识逐渐深入。软件工作的范围从只是使用程序设计语言编写程序，扩展到整个软件生存期。诸如，软件概念的形成、需求分析、设计、实现、测试、制造、安装和检验、运行和维护直到软件引退（为新的软件所代替）。同时还有许多技术管理工作（如过程管理、产品管理、资源管理）以及确认与验证工作（如评审与审计、产品分析、测试等）常常是跨越软件生存期各个阶段的专门工作。所有这些方面都应逐步建立起标准或规范来。

另一方面，软件工程标准的类型也是多方面的。它可能包括过程标准（如方法、技术、度量等）、产品标准（如需求、设计、部件、描述、计划、报告等）、专业标准（如职别、道德准则、认证、特许、课

程等)以及记法标准(如术语、表示法、语言等)。

表 9.1 (a)软件工程标准分类

			软件生存期								
			概念	需求	设计	实现	测试	制造	安装与检验	运行与维护	引退
标准类型	过程	方法									
		技术									
		度量									
	产品	需求									
		设计									
		部件									
		描述									
		计划									
		报告									
	专业	职别									
		道德准则									
		认证									
		特许									
		课程									
	记法	术语									
		表示法			ISO5807						
		语言									

在全面考虑以上两个方面的情况下，软件工程的标准可用一张二维的表格来表示。表 9.1(a)和(b)给出了这个二维表的大致格式。(b)表是(a)表的继续。表中填入了三个标准的例子(请注意它们在表中所处的位置：

① FIPS105 是美国国家标准局发布的《软件文档管理指南》(National Bureau of Standards, Guideline for Software Documentation Management, FIPS PUB 105, June 1984)

表 9.1　(b)软件工程标准分类

			技术管理			确认与验证		
			过程管理	产品管理	资源管理	评审与审计	产品分析	测试
标准类型	过程	方法				NSAC-39	NSAC-39	NSAC-39
		技术	FIPS 105					
		度量						
	产品	需求						
		设计						
		部件						
		描述						
		计划						
		报告						
	专业	职别						
		道德准则						
		认证						
		特许						
		课程						
	记法	术语						
		表示法						
		语言						

② NSAC-39 是美国核子安全分析中心发布的《安全参数显示系统的验证与确认》(Nuclear Safety Analysis Center, Verification and Validation for Safety Parameter Display Systems, NSAC-39, December1981)

③ ISO 5807 是国际标准化组织公布(现已成为我国国家标准)的《信息处理——数据流程图、程序流程图、系统流程图、程序网络图和系统资源图的文件编制符号及约定》(本书第四章 4.1 节讨论过的标准程序流程图正是以此为依据)。

这个表不仅告诉了我们软件工程标准的范围和标准如何分

类，而且对标准的开发具有指导作用。已经制定的标准都可在表中找到相应的位置，而且它可启发我们去制定新的标准。

9.2 软件工程标准化的意义

为什么要积极推行软件工程标准化工作，其道理是显而易见的。仅就一个软件开发项目来说，有多个层次、不同分工的人员相互配合，在开发项目的各个部分以及各开发阶段之间也都存在着许多联系和衔接问题。如何把这些错综复杂的关系协调好，需要有一系列统一的约束和规定。在软件开发项目取得阶段成果或最后完成时，需要进行阶段评审和验收测试。投入运行的软件，其维护工作中 遇到的问题又与开发工作有着密切的关系。软件的管理工作则渗透到软件生存期的每一个环节。所有这些都要求提供统一的行动规范和衡量准则，使得各种工作都能有章可循。

软件工程的标准化会给软件工作带来许多好处，比如：

- 提高软件的可靠性、可维护性和可移植性（这表明软件工程标准化可提高软件产品的质量）
- 提高软件的生产率
- 提高软件人员的技术水平
- 提高软件人员之间的通信效率，减少差错和误解
- 有利于软件管理
- 有利于降低软件产品的成本和运行维护成本
- 有利于缩短软件开发周期

9.3 软件工程标准的制定与推行

软件工程标准的制定与推行通常要经历一个环状的生命期（参看图 9.1）。最初，制定一项标准仅仅是初步设想，经发起后沿着环状生命期，顺时针进行要经历以下的步骤：

① 建议：拟订初步的建议方案

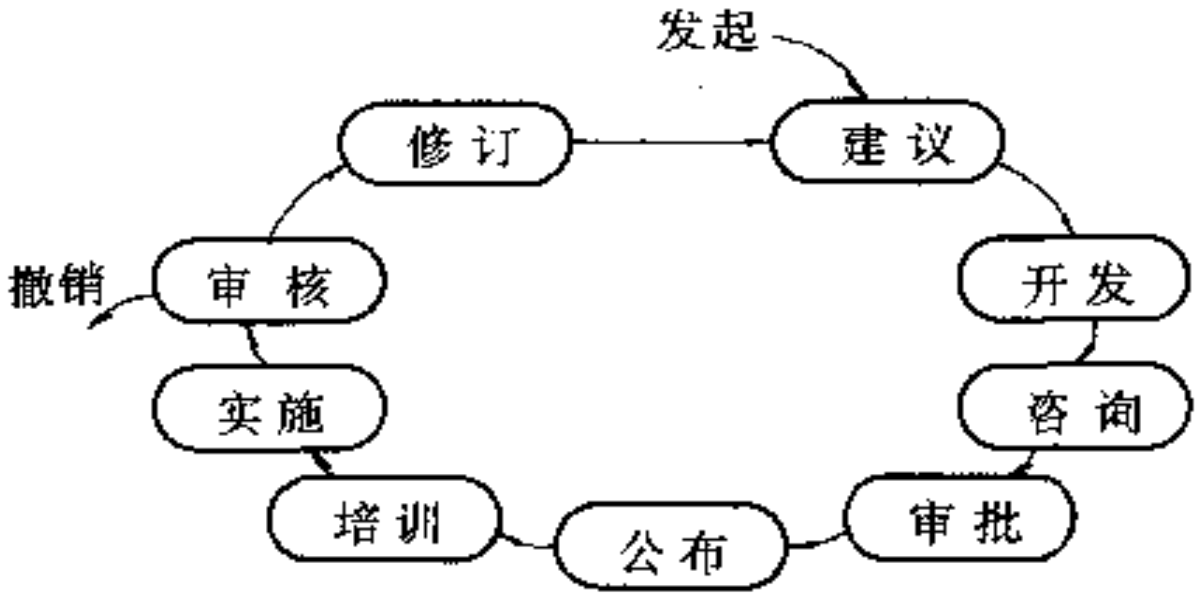


图 9.1　软件工程标准的环状生命期

② 开发：制定标准的具体内容

③ 咨询：征求并吸收有关人员意见

④ 审批：由管理部门决定能否推出

⑤ 公布：公开发布，使标准生效

⑥ 培训：为推行标准准备人员条件

⑦ 实施：投入使用，需经历相当期限

⑧ 审核：检验实施效果，决定修订还是撤销

⑨ 修订：修改其中不适当的部分，形成标准的新版本，进入新的周期

为使标准逐步成熟，可能在环状生命周期上循环若干圈，需要做大量的工作。事实上，软件工程标准在制定和推行过程中还会遇到许多实际问题。其中影响软件工程标准顺利实施的一些不利因素应当特别引起重视。这些因素可能有：

① 标准本身制定得有缺陷，或是存在不够合理，不够恰当的部分。

② 标准文本编写得有缺点，例如，文字叙述可读性差，难于理解，或是缺少实例供读者参阅。

③ 主管部门未能坚持大力推行，在实施的过程中遇到问题又

未能及时加以解决。

④ 未能及时作好宣传、培训和实施指导。

⑤ 未能及时修订和更新。

由于标准化的方向是无可置疑的，我们应该努力克服困难，排除各种障碍，坚定不移地推动软件工程标准化更快地发展。

9.4 软件工程标准的层次

根据软件工程标准制定的机构和标准适用的范围有所不同，它可分为五个级别，即国际标准、国家标准、行业标准、企业(机构)标准及项目(课题)标准。以下分别对五级标准的标识符及标准制定(或批准)的机构作一简要说明：

① 国际标准

由国际联合机构制定和公布，提供各国参考的标准。

• ISO(International Standards Organization)——国际标准化组织。这一国际机构有着广泛的代表性和权威性，它所公布的标准也有较大影响。60 年代初，该机构建立了“计算机与信息处理技术委员会”(简称 ISO/TC97)，专门负责与计算机有关的标准化工作。这一标准通常标有 ISO 字样，如 ISO 8631-86 Information processing -Program constructs and conventions for their representation (信息处理——程序构造及其表示法的约定。现已被我国收入国家标准)。

② 国家标准

由政府或国家级的机构制定或批准，适用于全国范围的标准，如：

• GB——中华人民共和国国家技术监督局是我国的最高标准化机构，它所公布实施的标准简称为“国标”。现已批准了若干个软件工程标准(详见本章 9.5 节)。

• ANSI(American National Standards Institute)——美国国家标准协会。这是美国一些民间标准化组织的领导机构，具有一定权

威性。

· FIPS(NBS)〔Federal Information Processing Standards (National Bureau of Standards)〕——美国商务部国家标准局联邦信息处理标准。它所公布的标准均冠有 FIPS 字样，如，1987 年发表的 FIPS PUB 132-87 Guideline for validation and verification plan of computer software 软件确认与验证计划指南。

· BS(British Standard)——英国国家标准。

· JIS(Japanese Industrial Standard)——日本工业标准。

③ 行业标准

由行业机构、学术团体或国防机构制定，并适用于某个业务领域的标准，如：

· IEEE(Institute of Electrical and Electronics Engineers)——美国电气和电子工程师学会。近年该学会专门成立了软件标准分技术委员会(SESS)，积极开展了软件标准化活动，取得了显著成果，受到了软件界的关注。IEEE 通过的标准常常要报请 ANSI 审批，使其具有国家标准的性质。因此，我们看到 IEEE 公布的标准常冠有 ANSI 字头。例如，ANSI/IEEE Str 828-1983 软件配置管理计划标准。

· GJB——中华人民共和国国家军用标准。这是由我国国防科学技术工业委员会批准，适合于国防部门和军队使用的标准。例如，1988 年发布实施的 GJB473-88 军用软件开发规范。

· DOD-STD(Department Of Defense-STanDards)——美国国防部标准。适用于美国国防部门。

· MIL-S(MILitary-Standards)——美国军用标准。适用于美军内部。

此外，近年来我国许多经济部门(例如，航天航空部、原国家机械工业委员会、对外经济贸易部、石油化学工业总公司等。)开展了软件标准化工作，制定和公布了一些适应于本部门工作需要的

规范。这些规范大都参考了国际标准或国家标准，对各自行业所属企业的软件工程工作起了有力的推动作用。

④ 企业规范

一些大型企业或公司，由于软件工程工作的需要，制定适用于本部门的规范。例如，美国 IBM 公司通用产品部(General Products Division)1984 年制定的“程序设计开发指南”，仅供该公司内部使用。

⑤ 项目规范

由某一科研生产项目组织制定，且为该项任务专用的软件工程规范。例如，计算机集成制造系统(CIMS)的软件工程规范。

9.5 我国的软件工程标准化工作

1983 年 5 月我国国家标准总局和原电子工业部主持成立了“计算机与信息处理标准化技术委员会”，下设十三个分技术委员会。和软件相关的是程序设计语言分技术委员会和软件工程技术委员会。我国制定和推行标准化工作的总原则是向国际标准靠拢，对于能够在我国适用的标准一律按等同采用的方法，以促进国际交流。

现已得到国家标准总局批准或即将获得批准的软件工程国家标准有：

- 软件开发规范 GB 8566-88
- 软件产品开发文件编制指南 GB 8567-88
- 计算机软件需求规格说明编制指南 GB 9385-88
- 计算机软件测试文件编制规范 GB 9386-88
- 软件工程术语标准 GB/T 11457-89
- 信息处理——数据流程图、程序流程图、系统流程图、程序网络图和系统资源图的文件编制符号及约定 GB 1526-89

除此以外，还有一批国家标准正在起草中，同时国防科工委

组织制定了一套“军标”，各部委也正在制定和实施适用于本行业领域的标准或规范。总的说来，软件工程标准化工作仍处于起步阶段，它在提高我国软件工程水平，促进我国软件产业的发展以及加强和国外的软件交流等方面必将起到应有的作用。

9.6 文档的作用和分类

软件文档(document)也称文件，通常指的是一些记录的数据和数据媒体，它具有固定不变的形式，可被人和计算机阅读。它和计算机程序共同构成了能完成特定功能的计算机软件(有人把源程序也当作文档的一部分)。我们知道，硬件产品和产品资料在整个生产过程中都是有形可见的，软件生产则有很大不同，文档本身就是软件产品。没有文档的软件，不成其为软件，更谈不到软件产品。软件文档的编制(documentation)在软件开发工作中占有突出的地位和相当的工作量。高效率、高质量地开发、分发、管理和维护文档对于转让、变更、修正、扩充和使用文档，对于充分发挥软件产品的效益有着重要意义。

然而，在实际工作中，文档在编制和使用中存在着许多问题，有待于解决。软件开发人员中较普遍地存在着对编制文档不感兴趣的现象。从用户方面看，他们又常常抱怨：文档售价太高、文档不够完整、文档编写得不好、文档已经陈旧或是文档太多，难于使用等等。究竟应该怎样要求它，文档应该写哪些，说明什么问题，起什么作用？这里将给出简要的介绍。

文档在软件开发人员、软件管理人员、维护人员、用户以及计算机之间的多种桥梁作用可从图 9.2 中看出。软件开发人员在各个阶段中以文档作为前阶段工作成果的体现和后阶段工作的依据，这个作用是显而易见的。软件开发过程中软件开发人员需制定一些工作计划或工作报告，这些计划和报告都要提供给管理人员，并得到必要的支持。管理人员则可通过这些文档了解软件开发项

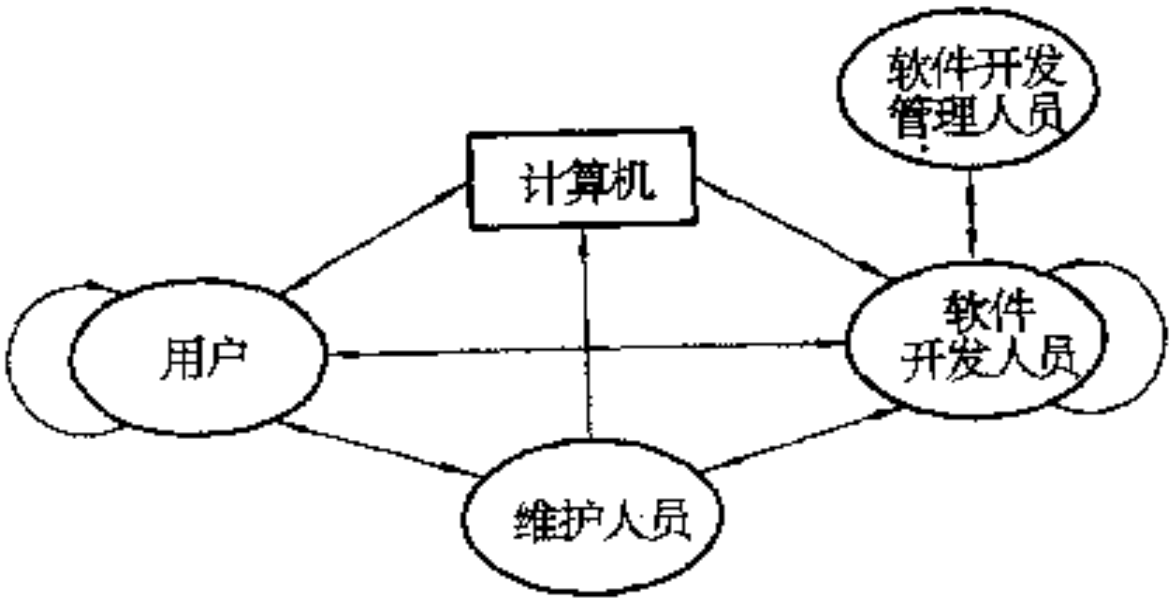


图 9.2　文档桥梁作用

目安排、进度、资源使用和成果等。软件开发人员需为用户了解软件的使用、操作和维护提供详细的资料，我们称此为用户文档。以上三种文档构成了软件文档的主要部分。我们把这三种文档所包括的内容列在图 9.3 中。其中列举了十三个文档，这里对它们作一些简要说明：

• 可行性研究报告：说明该软件开发项目的实现在技术上、经济上和社会因素上的可行性，评述为了合理地达到开发目标可供选择的各种可能实施的方案，说明并论证所选定实施方案的理由。

• 项目开发计划：为软件项目实施方案制定出具体计划，应该包括各部分工作的负责人员、开发的进度、开发经费的预算、所需的硬件及软件资源等。项目开发计划应提供给管理部门，并作为开发阶段评审的参考。

• 软件需求说明书：也称软件规格说明书，其中对所开发软件的功能、性能、用户界面及运行环境等作出详细的说明。它是用户与开发人员双方对软件需求取得共同理解基础上达成的协议，也是实施开发工作的基础。

• 数据要求说明书：该说明书应给出数据逻辑描述和数据采

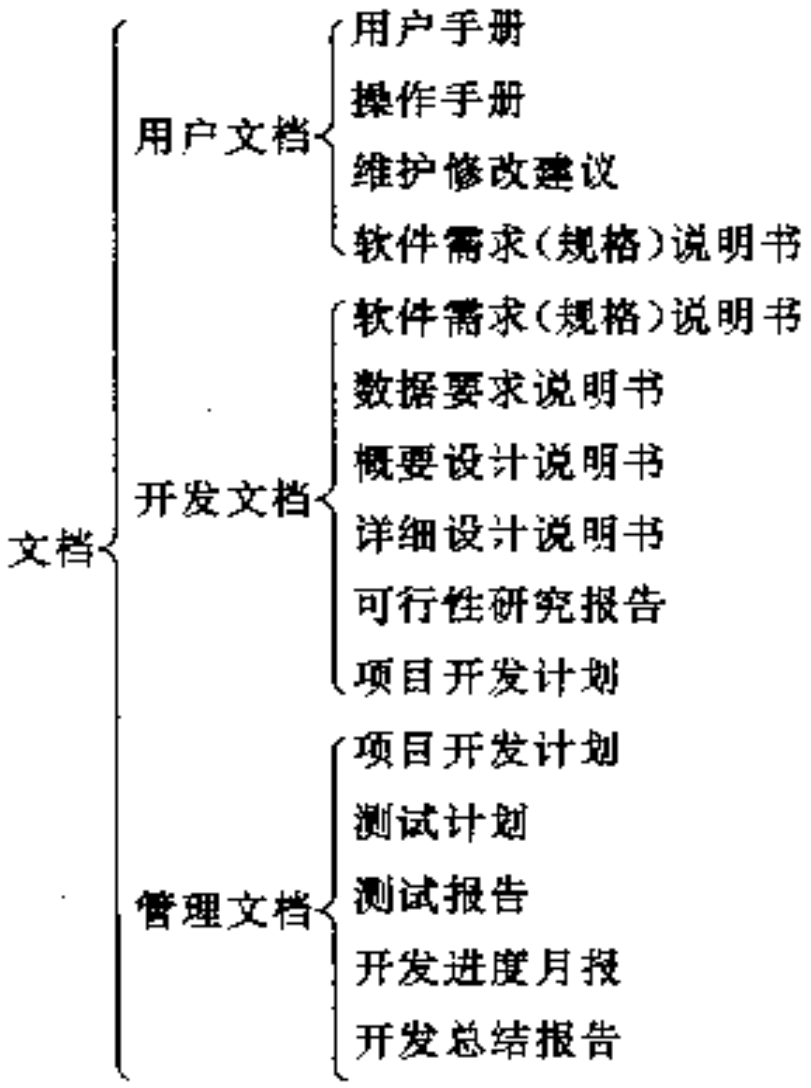


图 9.3　三种文档

集的各项要求，为生成和维护系统数据文卷作好准备。

• 概要设计说明书：该说明书是概要设计阶段的工作成果，它应说明功能分配、模块划分、程序的总体结构、输入输出以及接口设计、运行设计、数据结构设计和出错处理设计等，为详细设计奠定基础。

• 详细设计说明书：着重描述每一模块是怎样实现的，包括实现算法、逻辑流程等。

• 用户手册：本手册详细描述软件的功能、性能和用户界面，使用户了解如何使用该软件。

• 操作手册：本手册为操作人员提供该软件各种运行情况的有关知识，特别是操作方法的具体细节。

• 测试计划：为做好组装测试和确认测试，需为如何组织测试制定实施计划。计划应包括测试的内容、进度、条件、人员、测试用例的选取原则、测试结果允许的偏差范围等。

• 测试分析报告：测试工作完成以后，应提交测试计划执行情况的说明。对测试结果加以分析，并提出测试的结论意见。

• 开发进度月报：该月报系软件人员按月向管理部门提交的项目进展情况报告。报告应包括进度计划与实际执行情况的比较、阶段成果、遇到的问题和解决的办法以及下个月的打算等。

• 项目开发总结报告：软件项目开发完成以后，应与项目实施计划对照，总结实际执行的情况，如进度、成果、资源利用、成本

和投入的人力。此外还需对开发工作作出评价，总结出经验和教训。

• 维护修改建议：软件产品投入运行以后，发现了需对其进行修正、更改等问题，应将存在的问题、修改的考虑以及修改的影响估计作详细的描述，写成维护修改建议，提交审批。

以上这些文档是在软件生存期中，随着各阶段工作的开展适时编制。其中有的仅反映一个阶段的工作，有的则需跨越多个阶段。表 9.2 给出了各个文档应在软件生存期中哪个阶段编写。这些文档最终要向软件管理部门，或是向用户回答以下的问题：

表9.2 软件生存期各阶段编制的文档

文档 \ 阶段	可行性研究与计划	需求分析	设计	代码编写	测试	运行与维护
可行性研究报告	→					
项目开发计划	→	→				
软件需求说明		→				
数据要求说明		→				
概要设计说明			→			
详细设计说明			→			
测试计划		→	→			
用户手册		→	→	→		
操作手册			→	→		
测试分析报告					→	
开发进度月报	→	→	→	→	→	
项目开发总结					→	
维护修改建议						→

• 哪些需求要被满足，即回答“做什么？”

• 所开发的软件在什么环境中实现以及所需信息从哪里来，

即回答“从何处?”

- 某些开发工作的时间如何安排,即回答“何时干?”
- 某些开发(或维护)工作打算由“谁来干?”
- 某些需求是怎么实现的?
- 为什么要进行那些软件开发或维护修改工作?

上述十三个文档都在一定程度上回答了这六个方面的问题。这可从表 9.3 中看到。

表 9.3　文档所回答的问题

所提问题 / 文档	什么	何处	何时	谁	如何	为何
可行性研究报告	√					√
项目开发计划	√		√	√		
软件需求说明	√	√				
数据要求说明	√	√				
概要设计说明					√	
详细设计说明					√	
测试计划			√	√	√	
用户手册					√	
操作手册					√	
测试分析报告	√					
开发进度月报	√		√			
项目开发总结	√					
维护修改建议	√			√		√

至此,我们对文档的作用有了进一步的理解。每一个文档的任

务也是明确的，任何一个文档都此是多余的。

9.7 文档编制的质量要求

为了使软件文档能起到前节所提到的多种桥梁作用，使它有助于程序员编制程序，有助于管理人员监督和管理软件开发，有助于用户了解软件的工作和应做的操作，有助于维护人员进行有效的修改和扩充，文档的编制必须保证一定的质量。质量差的软件文档不仅使读者难于理解，给使用者造成许多不便，而且会削弱对软件的管理（管理人员难以确认和评价开发工作的进展），增高软件的成本（一些工作可能被迫返工），甚至造成更加有害的后果（如误操作等）。

造成软件文档质量不高的原因可能是：

· 缺乏实践经验，缺乏评价文档质量的标准。

· 不重视文档编写工作或是对文档编写工作的安排不恰当。最常见到的情况是，软件开发过程中不能按表 9.2 给出的进度，分阶段及时完成文档的编制工作，而是在开发工作接近完成时集中人力和时间专门编写文档。另一方面，和程序工作相比，许多人对编制文档不感兴趣。于是在程序工作完成以后，不得不应付一下，把要求提供的文档赶写出来。这样的做法不可能得到高质量的文档。实际上，要得到真正高质量的文档并不容易，除去应在认识上对文档工作给予足够的重视外，常常需要经过编写初稿，听取意见进行修改，甚至要经过重新改写的过程。

高质量的文档应当体现在以下一些方面：

① 针对性：文档编制以前应分清读者对象，按不同的类型、不同层次的读者，决定怎样适应他们的需要。例如，管理文档主要是面向管理人员的，用户文档主要是面向用户的，这两类文档不应像开发文档（面向软件开发人员）那样过多地使用软件的专业术语。

② 精确性：文档的行文应当十分确切，不能出现多义性的描述。同一课题若干文档内容应该协调一致，应是没矛盾的。

③ 清晰性：文档编写应力求简明，如有可能，配以适当的图表，以增强其清晰性。

④ 完整性：任何一个文档都应当是完整的、独立的，它应自成体系。例如，前言部分应作一般性介绍，正文给出中心内容，必要时还有附录，列出参考资料等。同一课题的几个文档之间可能有些部分相同，这些重复是必要的。例如，同一项目的用户手册和操作手册中关于本项目功能、性能、实现环境等方面的描述是没有差别的。特别要避免在文档中出现转引其它文档内容的情况。比如，一些段落并未具体描述，而用“见××文档××节”的方式，这将给读者带来许多不便。

⑤ 灵活性：各个不同的软件项目，其规模和复杂程度有着许多实际差别，不能一律看待。本章图 9.3 所列文档是针对中等规模的软件而言的。对于较小的或比较简单的项目，可做适当调整或合并。比如，可将用户手册和操作手册合并成用户操作手册；软件需求说明书可包括对数据的要求，从而去掉数据要求说明书；概要设计说明书与详细设计说明书合并成软件设计说明书等。

⑥ 可追溯性：由于各开发阶段编制的文档与各阶段完成的工作有着紧密的关系，前后两个阶段生成的文档，随着开发工作的逐步扩展，具有一定的继承关系。在一个项目各开发阶段之间提供的文档必定存在着可追溯的关系。例如，某一项软件需求，必定在设计说明书，测试计划以至用户手册中有所体现。必要时应能做到跟踪追查。

9.8 文档的管理和维护

在整个软件生存期中，各种文档作为半成品或是最终成品，会不断地生成、修改或补充。为了最终得到高质量的产品，达到上

节提出的质量要求，必须加强对文档的管理。以下几个方面是应注意做到的：

① 软件开发小组应设一位文档保管人员，负责集中保管本项目已有文档的两套主文本。两套文本内容完全一致。其中的一套可按一定手续，办理借阅。

②软件开发小组的成员可根据工作需要在自己手中保存一些个人文档。这些一般都应是主文本的复制件，并注意和主文本保持一致，在作必要的修改时，也应先修改主文本。

③ 开发人员个人只保存着主文本中与他工作相关的部分文档。

④ 在新文档取代了旧文档时，管理人员应及时注销旧文档。在文档内容有更动时，管理人员应随时修订主文本，使其及时反映更新了的内容。

⑤ 项目开发结束时，文档管理人员应收回开发人员的个人文档。发现个人文档与主文本有差别时，应立即着手解决。这常常是未及时修订主文本造成的。

⑥ 在软件开发过程中，可能发现需要修改已完成的文档，特别是规模较大的项目，主文本的修改必须特别谨慎。修改以前要充分估计修改可能带来的影响，并且要按照：提议、评议、审核、批准和实施等步骤加以严格的控制。

附录

文档编写纲要

为使读者具体了解怎样编写文档，这里列出了十种文档的内容要求及其简要说明。这些文档包括：

1. 可行性研究报告
2. 项目开发计划
3. 需求规格说明书
4. 概要设计说明书
5. 详细设计说明书
6. 用户操作手册
7. 测试计划
8. 测试报告
9. 开发进度月报
10. 项目开发总结报告

各文档内容大纲由带编号的标题构成，标题后方括号内为其说明。

这里给出一个统一的文档封面格式：

文档编号_____

版本号 _____

文档名称 :_____

项目名称 :_____

项目负责人:_____

编写_____ ____年__月__日

校对_____ ____年__月__日

审核_____ ____年__月__日

批准_____ ____年__月__日

开发单位_____

一、 可行性研究报告

1. 引言

1.1 编写目的 〔阐明编写本可行性研究报告的目的，指出读者对象。〕

1.2 项目背景 〔应包括：a. 所建议开发的软件名称；b. 本项目的任务提出者、开发者及用户；c. 本项目与其它软件或其它系统的关系。〕

1.3 定义 〔列出本文档中用到的专门术语的定义和缩写词的原文。〕

1.4 参考资料 〔列出有关资料的作者、标题、编号、发表日期、出版单位或资料来源，可包括：a. 本项目经核准的计划任务书、合同或上级机关的批文；b. 与本项目有关的已发表的资料；c. 本文档中所引用的资料，所采用的软件标准或规范。〕

2. 可行性研究的前提

2.1 要求 〔列举并说明建议开发软件的基本要求，如 a. 功能；b. 性能；c. 输出；d. 输入；e. 基本的数据流程和处理流程；f. 安全与保密要求；g. 与本软件相关的其它系统；h. 完成期限。〕

2.2 目标 〔可包括：a. 人力与设备费用的节省；b. 处理速度的提高；c. 控制精度或生产能力的提高；d. 管理信息服务的改进；e. 决策系统的改进；f. 人员工作效率的提高，等等。〕

2.3 条件、假定和限制 〔可包括：a. 建议开发软件运行的最短寿命；b. 进行系统方案选择比较的期限；c. 经费来源和使用限制；d. 法律和政策方面的限制；e. 硬件、软件、运行环境和开发环境的条件和限制；f. 可利用的信息和资源；g. 建议开发软件投入使用的最迟时间。〕

2.4 可行性研究方法

2.5 决定可行性的主要因素

3. 对现有系统的分析

3.1 处理流程和数据流程

3.2 工作负荷

3.3 费用支出 〔如人力、设备、空间、支持性服务、材料等项开支。〕

3.4 人员 〔列出所需人员的专业技术类别和数量。〕

3.5 设备

3.6 局限性 〔说明现有系统存在的问题以及为什么需要开发新系统。〕

4. 所建议系统技术可行性分析

4.1 对系统的简要描述

4.2 处理流程和数据流程

4.3 与现有系统比较的优越性

4.4 采用建议系统可能带来的影响

4.4.1 对设备的影响

4.4.2 对现有软件的影响

4.4.3 对用户的影响

4.4.4 对系统运行的影响

4.4.5 对开发环境的影响

4.4.6 对运行环境的影响

4.4.7 对经费支出的影响

4.5 技术可行性评价 〔包括:a.在限制条件下,功能目标能否达到;b.利用现有技术,功能目标能否达到;c.对开发人员数量和质量的要求,并说明能否满足;d.在规定的期限内,开发能否完成。〕

5. 所建议系统经济可行性分析

5.1 支出

5.1.1 基建投资

5.1.2 其它一次性支出

5.1.3 经常性支出

5.2 效益

5.2.1 一次性收益

5.2.2 经常性收益

5.2.3 不可定量收益

5.3 收益/投资比

5.4 投资回收周期

5.5 敏感性分析 〔敏感性分析是指一些关键性因素,如:系统生存周期长度、系统工作负荷量、处理速度要求、设备和软件配置变化对支出和效益的影响等的分析。〕

6. 社会因素可行性分析

6.1 法律因素 〔如，合同责任、侵犯专利权、侵犯版权等问题的分析。〕

6.2 用户使用可行性 〔如，用户单位的行政管理、工作制度、人员素质等能否满足要求。〕

7. 其它可供选择的方案 〔逐个阐明其它可供选择的方案，并重点说明未被推荐的理由。〕

8. 结论意见 〔结论意见可能是：a. 可着手组织开发；b. 需待若干条件（如资金、人力、设备等）具备后才能开发；c. 需对开发目标进行某些修改；d. 不能进行或不必进行（如技术不成熟，经济上不合算等）；e. 其它。〕

二、 项目开发计划

1. 引言

1.1 编写目的 〔阐明编写本开发计划的目的，指出读者对象。〕

1.2 项目背景 〔可包括：a. 本项目的委托单位、开发单位及主管部门；b. 该软件系统与其它系统的关系。〕

1.3 定义 〔列出本文档中用到的专门术语的定义和缩写词的原文。〕

1.4 参考资料 〔可包括：a. 本项目经核准的计划任务书、合同或上级机关的批文；b. 本文档所引用的资料、规范等。列出这些资料的作者、标题、编号、发表日期、出版单位或资料来源。〕

2. 项目概述

2.1 工作内容 〔简要说明本项目的各项主要工作，介绍所开发软件的功能、性能等。若不编写可行性研究报告，则应在本节给出较详细的介绍。〕

2.2 条件与限制 〔阐明为完成本项目应具备的条件、开发单位已具备的条件以及尚需创造的条件。必要时还应说明用户及分合同承包者承担的工作、完成期限及其它条件与限制。〕

2.3 产品

2.3.1 程序 〔列出应交付的程序名称、使用的语言及存储形式。〕

2.3.2 文档 〔列出应交付的文档。〕

2.4 运行环境 〔应包括硬件环境、软件环境。〕

2.5 服务 〔阐明开发单位可向用户提供的服务，如人员培训、安装、保修、维护和其它运行支持。〕

2.6 验收标准

3. 实施计划

3.1 任务分解 〔任务的划分及各项任务的负责人。〕

3.2 进度 〔按阶段完成的项目，用图表说明开始时间、完成时间。〕

3.3 预算

3.4 关键问题 〔说明可能影响项目成败的关键问题，如设备条件、技术难点或其它风险因素，并说明对策。〕

4. 人员组织及分工

5. 交付期限

6. 专题计划要点 〔如测试计划、质量保证计划、配置管理计划、人员培训计划、系统安装计划等。〕

三、 需求规格说明书

1. 引言

1.1 编写目的 〔阐明编写本需求规格说明书的目的，指明读者对象。〕

1.2 项目背景 〔应包括：a. 本项目的委托单位、开发单位及主管部门；b. 该软件系统与其它系统的关系。〕

1.3 定义 〔列出本文档中所用到的专门术语的定义和缩写词的原文。〕

1.4 参考资料 〔可包括：a. 本项目经核准的计划任务书、合同或上级机关的批文；b. 项目开发计划；c. 本文档所引用的资料、标准和规范。列出这些资料的作者、标题、编号、发表日期、出版单位或资料来源。〕

2. 任务概述

2.1 目标

2.2 运行环境

2.3 条件与限制

3. 数据描述

3.1 静态数据

3.2 动态数据 〔包括输入数据和输出数据。〕

3.3 数据库描述 〔给出使用数据库的名称和类型。〕

3.4 数据词典

3.5 数据采集

4. 功能需求

4.1 功能划分

4.2 功能描述

5. 性能需求

5.1 数据精确度

5.2 时间特性 〔如响应时间、更新处理时间、数据转换与传输时间、运行时间等。〕

5.3 适应性 〔在操作方式、运行环境、与其它软件的接口以及开发计划等发生变化时，应具有的适应能力。〕

6. 运行需求

6.1 用户界面 〔如屏幕格式、报表格式、菜单格式、输入输出时间等。〕

6.2 硬件接口

6.3 软件接口

6.4 故障处理

7. 其它需求 〔如可使用性、安全保密、可维护性、可移植性等。〕

四、 概要设计说明书

1. 引言

1.1 编写目的 〔阐明编写本概要设计说明书的目的，指明读者对象。〕

1.2 项目背景 〔可包括：a. 本项目的委托单位、开发单位及上级管理部门；b. 该软件系统与其它系统的关系。〕

1.3 定义 〔列出本文档中用到的专门术语的定义和缩写词的原意。〕

1.4 参考资料 〔列出有关资料的作者、标题、编号、发表日期、出版单位或资料来源，可包括：a. 本项目经核准的计划任务书，合同或上级机关的批文；b. 项目开发计划；c. 需求规格说明书；d. 测试计划(初稿)；e. 用户操作手册(初稿)；f. 本文档所引用的资料、采用的标准或规范。〕

2. 任务概述

2.1 目标

2.2 运行环境

2.3 需求概述

2.4 条件与限制

3. 总体设计

3.1 处理流程

3.2 总体结构和模块外部设计

3.3 功能分配 〔表明各项功能与程序结构的关系〕

4. 接口设计

4.1 外部接口 〔包括用户界面、软件接口和硬件接口。〕

4.2 内部接口 〔模块间的接口〕

5. 数据结构设计

5.1 逻辑结构设计

5.2 物理结构设计

5.3 数据结构与程序的关系

6. 运行设计

6.1 运行模块的组合

6.2 运行控制

6.3 运行时间

7. 出错处理设计

7.1 出错输出信息

7.2 出错处理对策 〔如设置后备、性能降级、恢复及再启动等。〕

8. 安全保密设计

9. 维护设计 〔说明为方便维护工作的设施，如维护模块等。〕

五、 详细设计说明书

1. 引言

1.1 编写目的 〔编写本设计说明书的目的及读者对象。〕

1.2 项目背景 〔项目来源及主管部门等。〕

1.3 定义 〔列出本文档中用到的专门术语的定义和缩写词的原意。〕

1.4 参考资料 〔列出有关资料的作者、标题、编号、发表日期、出版单位或资料来源，可包括：a. 本项目计划任务书、合同或批文；b. 项目开发计划；c. 需

求规格说明书;d. 概要设计说明书;e. 测试计划(初稿);f. 用户操作手册(初稿);g. 本文档中引用的其它资料、软件开发标准或规范。〕

2. 总体设计

2.1 需求概述

2.2 软件结构 〔如给出结构图。〕

3. 程序描述 〔逐个模块给出以下说明:〕

3.1 功能

3.2 性能

3.3 输入项

3.4 输出项

3.5 算法 〔本模块所选用的算法。〕

3.6 程序逻辑 〔详细描述本模块实现的算法,可采用:a. 标准流程图;b. PDL 语言;c. N-S 图;d. PAD;e. 判定表等描述算法的图表。〕

3.7 接口 〔给出本模块在结构图中与其上级模块及下属模块的调用关系及传递的信息。〕

3.8 存储分配

3.9 限制条件

3.10 测试要点 〔给出测试本模块的主要测试要求。〕

六、 用户操作手册

1. 引言

1.1 编写目的 〔阐明编写本手册的目的并指明读者对象。〕

1.2 项目背景 〔说明项目来源、委托单位、开发单位及主管部门。〕

1.3 定义 〔列出本手册中使用的专门术语的定义和缩写词的原意。〕

1.4 参考资料 〔列出有关资料的作者、标题、编号、发表日期、出版单位及资料来源,可包括:a. 本项目计划任务书、合同或批文;b. 项目开发计划;c. 需求规格说明书;d. 概要设计说明书;e. 详细设计说明书;f. 测试计划;g. 本手册中引用的其它资料。软件工程规范或软件工程标准。〕

2. 软件概述

2.1 目标

2.2 功能

2.3 性能

a. 数据精确度 〔包括输入、输出及处理数据的精度。〕

b. 时间特性 〔如响应时间、处理时间、数据传输时间等。〕

c. 灵活性 〔在操作方式、运行环境等需作某些变更时，本软件的适应能力。〕

3. 运行环境

3.1 硬件 〔列出本软件系统运行所需的硬件最小配置，如：a. 计算机型号、主存容量；b. 外存储器、媒体、记录格式、设备型号及数量；c. 输入、输出设备；d. 数据传输设备及数据转换设备的型号和数量。〕

3.2 支持软件 〔如：a. 操作系统名称及版本号；b. 语言编译系统或汇编系统的名称及版本号；c. 数据库管理系统的名称及版本号；d. 其它必要的支持软件。〕

4. 使用说明

4.1 安装和初始化 〔给出程序的存储形式、操作命令、反馈信息及其含意、表明安装完成的测试实例以及安装所需的软件工具等。〕

4.2 输入 〔给出输入数据或参数的要求。〕

4.2.1 数据背景 〔说明数据来源、存储媒体、出现频度、限制和质量管理等。〕

4.2.2 数据格式 〔如：a. 长度；b. 格式基准；c. 标号；d. 顺序；e. 分隔符；f. 词汇表；g. 省略和重复；h. 控制。〕

4.2.3 输入举例

4.3 输出 〔给出每项输出数据的说明。〕

4.3.1 数据背景 〔说明输出数据的去向、使用频度、存放媒体及质量管理等。〕

4.3.2 数据格式 〔详细阐明每一输出数据的格式，如：首部、主体和尾部的具体形式。〕

4.3.3 举例

4.4 出错和恢复 〔给出：a. 出借信息及其含意；b. 用户应采取的措施，如修改、恢复、再启动等。〕

4.5 求助查询 〔说明如何操作用户可得到指示。〕

5. 运行说明

5.1 运行表 〔列出每种可能的运行情况，说明其运行目的。〕

5.2 运行步骤 〔按顺序说明每种运行的步骤应包括：〕

5.2.1 运行控制

5.2.2 操作信息

a. 运行目的；b. 操作要求；c. 启动方法；d. 预计运行时间；e. 操作命令格式及说明；f. 其它事项。

5.2.3 输入/输出文件 〔给出建立或更新文件的有关信息如：〕

a. 文件的名称及编号；b. 记录媒体；c. 存留的目录；d. 文件的支配〔说明确定保留文件或废弃文件的准则，分发文件的对象，占用硬件的优先级及保密控制等。〕

5.2.4 启动或恢复过程

6. 非常规过程 〔提供应急或非常规操作的必要信息及操作步骤，如出错处理操作、向后备系统切换操作以及维护人员须知的操作和注意事项。〕

7. 操作命令一览表 〔按字母顺序逐个列出全部操作命令的格式、功能及参数说明等。〕

8. 程序文件(或命令文件)和数据文件一览表 〔按文件名字母顺序或按功能与模块分类顺序逐个列出文件名称、标识符及说明。〕

9. 用户操作举例

七、 测试计划

1. 引言

1.1 编写目的 〔阐明编写本测试计划的目的并指明读者对象。〕

1.2 项目背景 〔说明项目来源、委托单位、开发单位及主管理部门。〕

1.3 定义 〔列出本测试计划中用到的专门术语的定义和缩写词的原意。〕

1.4 参考资料 〔列出有关资料的作者、标题、编号、发表日期、出版单位或资料来源，可包括：a. 本项目计划任务书、合同或批文；b. 项目开发计划；c. 需求规格说明书；d. 概要设计说明书；e. 详细设计说明书；f. 用户操作手册；g. 本测试计划中引用的其它资料、软件开发标准或规范。〕

2. 任务概述

2.1 目标

2.2 运行环境

2.3 需求概述

2.4 条件与限制

3. 计划

3.1 测试方案 〔说明确定测试方法和选取测试用例的原则。〕

3.2 测试项目 〔列出组装测试和确认测试中每一项测试的内容、名称、目的和进度。〕

3.2 测试准备

3.3 测试机构及人员 〔机构名称、负责人和职责。〕

4. 测试项目说明 〔按顺序对逐个测试项说明。〕

4.1 测试项名称及测试内容

4.2 测试用例

4.2.1 输入 〔输入的数据和输入命令。〕

4.2.2 输出 〔预期的输出数据。〕

4.2.3 步骤及操作

4.2.4 允许偏差 〔给出测试结果与预期结果允许偏差的范围。〕

4.3 进度

4.4 条件 〔给出本项测试对资源的特殊要求，如设备、软件、人员等。〕

4.5 测试资料 〔说明本项测试所需的资料。〕

5. 评价

5.1 范围 〔说明所完成的各项测试说明问题的范围及其局限性。〕

5.2 准则 〔说明评价测试结果的准则。〕

八、 测试分析报告

1. 引言

1.1 编写目的 〔阐明编写本测试分析报告的目的并指明读者对象。〕

1.2 项目背景 〔说明项目来源、委托单位、开发单位及主管部门〕

1.3 定义 〔列出本测试分析报告中用到的专门术语的定义和缩写词的原意。〕

1.4 参考资料 〔列出有关资料的作者、标题、编号、发表日期、出版单位或资料来源，可包括：a. 本项目计划任务书、合同或批文；b. 项目开发计划；c. 需求规格说明书；d. 概要设计说明书；e. 详细设计说明书；f. 用户操作手册；g. 测试计划；h. 本测试分析报告中引用的其它资料、软件工程规范或软件工程标准。〕

2. 测试计划执行情况

2.1 测试项目 〔列出每一测试项目的名称、内容和目的。〕

2.2 测试机构及人员 〔给出测试机构名称、负责人及参与测试人员名单。〕

2.3 测试结果 〔按顺序给出每一测试项目的：a. 实测结果数据；b. 与预期结果数据的偏差；c. 该项测试表明的事实；d. 该项测试发现的问题。〕

3. 软件需求测试结论 〔按顺序给出每一项需求测试的结论，包括：a. 证实的软件能力；b. 局限性(即本项需求未得到充分测试的情况及原因)。〕

4. 评价

4.1 软件能力 〔经过测试所表明的软件能力。〕

4.2 缺陷和限制 〔说明测试所揭露的软件缺陷和不足以及可能给软件运行带来的影响。〕

4.3 建议 〔提出为弥补上述缺陷的建议。〕

4.4 测试结论 〔说明能否通过。〕

九、 开发进度月报

1. 报告时间及所处的开发阶段

2. 工程进度

2.1 本月内的主要活动

2.2 实际进展与计划比较

3. 所用工时 〔按不同层次人员分别计时。〕

4. 所用机时 〔按所用计算机机型分别计时。〕

5. 经费支出 〔分类列出本月经费支出项目，给出支出总额，并与计划比较。〕

6. 工作遇到的问题及采取的对策

7. 本月完成的成果

8. 下月工作计划

9. 特殊问题

十、 项目开发总结报告

1. 引言

1.1 目的 〔阐明编写本总结报告的目的，指明读者对象。〕

1.2 项目背景 〔说明项目来源、委托单位、开发单位及主管部门。〕

1.3 定义 〔列出本报告用到的专门术语及缩写词含意。〕

1.4 参考资料 〔列出有关资料的作者、标题、编号、发表日期、出版单位或资料来源，可包括：a. 本项目经核准的计划任务书、合同或上级机关的批文；b. 项目开发计划；c. 需求规格说明书；d. 概要设计说明书；e. 详细设计说明书；f. 用户操作手册；g. 测试计划；h. 测试分析报告；i. 本报告引用的其它资料、开发标准或开发规范等。〕

2. 开发结果

2.1 产品 〔可包括：a. 列出各部分的程序名称、源程序行数(包括注释行)或目标程序字节数及程序总计数量、存储形式；b. 产品文档名称等。〕

2.2 主要功能及性能

2.3 所用工时 〔按人员的不同层次分别计时。〕

2.4 所用机时 〔按所用计算机机型分别计时。〕

2.5 进度 〔给出计划进度与实际进度的对比。〕

2.6 费用

3. 评价

3.1 生产效率评价 〔如平均每人每月生产的源程序行数、文档的字数等。〕

3.2 技术方案评价

3.3 产品质量评价

4. 经验与教训

主要参考文献

1. N. D. Birrell and M. A. Ould: A Practical Handbook for Software Development, Cambridge University Press, 1985.

2. Glenford J. Myers: The Art of Software Testing, John Wiley & Sons, Inc., New York, 1979(中文译本:计算机软件测试技巧,周之英,郑人杰译,清华大学出版社,1985).

3. Brian W. Kernighan and P. J. Plauger: The Elements of Programming Style, Second Edition, McGraw-Hill Book Company, 1978(中文译本:程序设计技巧,晏晓焰译,清华大学出版社,1985).

4. Meilir Page-Jones: The Practical Guide to Structured Systems Design, Yourdon Inc., New York, 1980.

5. Joseph M. Fox: Software and its Development, Prentice-Hall Inc., Englewood Cliffs, N. J., 1982.

6. I. Sommerville: Software Engineering, Second Edition, Addison-Wesley Publishing Company, Wokingham, England, 1985.

7. R. S. Pressman: Software Engineering: A Practitioner's Approach, Second Edition, McGraw-Hill Book Company, New York, 1987.

8. Doug Bell, Ian Morrey and John Pugh: Software Engineering-A Programming Approach, Prentice-Hall International (UK) Ltd, London, 1987.

9. Richard Fairley: Software Engineering Concepts, McGraw-Hill, New York, 1985.

10. Barry W. Boehm: Software Engineering Economics, Prentice-Hall, Inc., Englewood Cliffs, N. J., 1981.

11. James Martin and Carma McClure: Software Maintenance, Prentice-Hall, Inc., 1983.

12. Software Systems Development Methodology Handbook, World Information Systems Enterprises, Inc., 1984.

13. Victor Weinberg: Structured Analysis, Prentice-Hall, Englewood Cliffs, N. J., 1978.

14. Tom DeMarco: Structured Analysis and System Specification, Yourdon

Inc. ,New York, 1979.

15. William S. Davis:Systems Analysis and Design-A Structured Approach, Addison-Wesley Publishing Company, 1983.

16. Donald J. Reifer:Tutorial-Software Management, IEEE Computer Society International Conference, San Francisco, 1979.

17. Wayne P. Stevens:Using Structured Design, John Wiley & Sons, New York, 1981.

18. Software Engineering Handbook, Prepared by General Electric Company, 1983(中文译本:软件工程指南,朱三元等编译,上海翻译出版公司,1985).

19. Edited by C. R. Vick and C. V. Ramamoorthy:Handbook of Software Engineering, Van Nostrand Reinhold Company Inc. ,1984.

20. F. P. Brooks:The Mythical Man-month, Addison-Wesley,1972.

21. 潘锦平,软件开发技术,上海科学技术文献出版社,1985.